Using Mobility and Exception Handling to Achieve Mobile Agents that Survive Server Crash Failures

Simon Pears

Ph.D Thesis

Department of Computer Science
University of Durham

April 2005

Abstract

Mobile agent technology, when designed and used effectively, can minimize bandwidth consumption and autonomously provide a snapshot of the current context of a distributed system. Protecting mobile agents from server crashes is a challenging issue, since developers normally have no control over remote servers. Server crash failures can leave replicas, in stable storage, unavailable for an unknown time period. Furthermore, few systems have considered the need for using a fault tolerant protocol among a group of collaborating mobile agents.

This thesis uses exception handling to protect mobile agents from server crash failures. An exception model is proposed for mobile agents and two exception handler designs are investigated. The first exists at the server that created the mobile agent and uses a timeout mechanism. The second, the mobile shadow scheme, migrates with the mobile agent and operates at the previous server visited by the mobile agent. A case study application has been developed to compare the performance of the two exception handler designs. Performance results demonstrate that although the second design is slower it offers the smaller trip time when handling a server crash. Furthermore, no modification of the server environment is necessary.

This thesis shows that the mobile shadow exception handling scheme reduces complexity for a group of mobile agents to survive server crashes. The scheme deploys a replica that monitors the server occupied by the master, at each stage of the itinerary. The replica exists at the previous server visited in the itinerary. Consequently, each group member is a single fault tolerant entity with respect to server crash failures. Other schemes introduce greater complexity and performance overheads since, for each stage of the itinerary, a group of replicas is sent to servers that offer an equivalent service. In addition, future research is established for fault tolerance in groups of collaborating mobile agents.

Table of Contents

List of Figures

List of Tables

Declaration

No material contained in this thesis has previously been submitted for a degree in this or any other university.

This research has been documented, in part, within the following publications:

- S.Pears, J.Xu and C.Boldyreff, "Mobile Agent Fault Tolerance for Information Retrieval Applications: An Exception Handling Approach," in *Proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS'03)*, Pisa, Italy, April, 2003, pp.115-124.
- S.Pears, J.Xu and C.Boldyreff, "A Dynamic Shadow Approach for Mobile Agents to Survive Crash Failures," in *Proceedings of the 6th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, Hokkaido, Japan, May, 2003, pp.113-120.

Statement of Copyright

Acknowledgements

There have been many significant events and experiences encountered during the completion of this thesis. These would not have been possible without the support and encouragement of a number of people.

Firstly, I would like to thank my supervisors: Professor Jie Xu, Professor Cornelia Boldyreff and Professor Keith Bennett for help, direction and advice. Their knowledge in the field of distributed systems has proved invaluable, both towards the completion of the thesis and for presentations at international conferences.

I also thank Paul Townend, Erica Yang, Nik Looker and other members of the Distributed Systems group for feedback and inspiration from seminars. Furthermore, thanks go to Paul, Phyo, David and Bob for the challenging games of snooker and demonstration of the 'super double' shot. One day I will get there!

I would also like to acknowledge Andrew Hatch and Phyo Kyaw for their company at the Finland summer school and for helping me survive my first flight! I also acknowledge the Japanese 'woman' and the taxi driver who helped me find my way when I was lost in Japan!

Thanks also go to my family who have been a constant source of support and encouragement during my time in academia. They have provided significant advice and stood by decisions that I have made. They are greatly appreciated. Also in memory of my gran, Minnie Russell.

Thank you to Patricia Richardson who had the 'privilege' of proof reading parts of this work.

Finally, thanks go to the Engineering and Physical Sciences Research Council (EPSRC) for funding this research.

Chapter 1 Introduction

1 Research aims

Waldo, in [Waldo01], implies the lack of a failure model for mobile agent systems:

"Finally, an agent system's implementation in the Jini technology model would provide a failure model that the agent community might find useful". [Waldo01]

The literature provides no consensus on what the terms *agent* and *mobile agent* mean. Indeed, there are two research communities for mobile agents, i.e. distributed systems and artificial intelligence. This thesis focuses on mobile agents that belong to the distributed systems community with the view that they are distributed objects with limited intelligence.

The thesis is interested in understanding mobile agents that fail due to an agent server crash. The literature has suggested many techniques for protecting the loss of mobile agents due to the crash of an agent server. However, there appear to be few implementations adopted in actual mobile agent systems. A frequently stated potential application domain for mobile agents is information retrieval. For example, mobile agents may be used to filter large quantities of information from remote hosts.

Some solutions [DeAssisSilva01, Mohindra00, Silva00, Strasser98] employ transaction processing to satisfy failure dependencies with agent servers, i.e. execution of a mobile agent modifies its internal state and the state of the agent server. However, for information retrieval applications, transaction processing solutions introduce unnecessary performance overheads since there are no state dependencies introduced between the mobile agent and remote agent servers. The mobile agent only consumes information at visited agent servers. Furthermore, there are some solutions that inject a replica into stable storage upon arrival at a host. However, in the event of an agent server crash, the replica is unavailable for an unknown time period.

This thesis is concerned with developing a framework that employs exception handling for mobile agents to survive crash failures of hosts visited on a trip. Consequently, no modification of the agent server environment is necessary. This increases the likelihood that the framework is interoperable between mobile agent systems. Furthermore, the application developer is free to elect how to handle the loss of a mobile agent through an agent server crash. A conceptual framework will provide the basis for understanding how mobile agents can survive the crash of hosts visited on a trip. This implies that an exception handling model is required to outline the components of a mobile agent system and suggest exception handling control flow.

To summarise, the aims of this research are:

1. The development of a conceptual model for exception handling in mobile agent systems.
2. The development of a conceptual framework to protect mobile agents from failure due to an agent server crash. The framework uses exception handling for mobile agents to survive crash failures of agent servers.
3. A conceptual framework for mobile agents to survive crash failures should be independent of the agent server environment. This facilitates implementation across all mobile agent systems.
4. The conceptual framework must have the potential to be adopted for a group of mobile agents. The opinion is that the conceptual framework should be independent from the group of mobile agents to reduce the complexity of the design.
5. It is hoped that the thesis will provide a better understanding of how exception handling can be used to protect mobile agents from host crashes.

2 Assumptions

- Mobile agents are assumed to belong to the distributed systems community. Consequently, in this thesis a mobile agent is an active object that can migrate autonomously between hosts to perform an application task on the behalf of a user.
- Mobile agents for information retrieval applications do not modify the state of visited hosts. Consequently, transaction processing at hosts is an unnecessary performance overhead for mobile agent information retrieval applications.
- A failure model is required that states the conditions of failure for the environment where mobile agents are situated.

3 Research method

The research method adopted for the thesis is an engineering one, based upon an iterative improvement of the conceptual exception handling framework. In particular, the framework is examined based upon the conceptual model of exception handling in mobile agent systems. Further consideration is given regarding how the framework can be adopted for groups of co-operating mobile agents. The exception handling framework is compared to other systems for mobile agents to survive crash failures. Potential is also considered regarding the adoption of the exception handling framework within a group of co-operating mobile agents for information retrieval applications.

4 Contribution of thesis

The main contribution of the thesis is an understanding of exception handling for mobile agent systems. Indeed, the literature has witnessed little work in this area. This is achieved by creating an exception handling model for mobile agent systems and then implementing an exception handling scheme to protect mobile agents from agent server crash failures. Subsequently, the case study implementation provides an insight into how the scheme compares with others for groups of mobile agents.

5 Criteria for success

The overall criteria for the success of this research may be considered to be the development and evaluation of an exception handling scheme to protect mobile agents against agent server crashes.

This may be broken down into a number of areas which will be addressed by the thesis. The criteria for success are therefore:

a) Create an exception handling model for mobile agent systems

Very few examples of exception handling for mobile agents exist. Therefore, before developing an exception handling scheme for mobile agents to survive agent server crash failures, it is necessary to identify how mobile agents interact with software services at remote agent servers. This aids in understanding the control flow of exceptions between the agent server and mobile agents. Furthermore, the unique aspects for exception handling in mobile agent systems can be considered.

b) Identify a failure model for mobile agent systems

With any fault tolerance design there is the necessity to outline a failure model. This provides an understanding of the likely ways in which mobile agent execution can fail. Only then can exception detection mechanisms and recovery prove to be effective.

c) Development of an exception handling scheme for the protection of mobile agents against agent server crashes

An exception handling scheme will be developed and evaluated to protect mobile agents from agent server crash failures. A prototype of the exception handling scheme will be implemented and deployed using a case study application. An experiment will be performed to investigate the trip time increase incurred when an agent server crash is encountered.

d) Identify the best approach for mobile agent groups to survive agent server crashes

Using the experience gained from the development of the exception handling scheme in c), the key aspects will be identified with respect to the suitability of the scheme for use with groups of collaborating mobile agents. Furthermore, consideration will be given towards the viability of the scheme developed in this thesis in comparison with other systems that protect mobile agents from agent server crash failures.

These criteria will be evaluated in chapter 6.

6 Thesis overview

This thesis is composed of seven chapters, of which this is the first. Chapter 2 provides an introduction to mobile agents. In particular, an overall introduction to the agent paradigm is provided and mobile agents are defined. The architecture of a typical mobile agent system is described before the chapter concludes, by identifying the current problems encountered with mobile agents.

Chapter 3 provides an introduction to exception handling. This highlights the difficulties and existing approaches for exception handling in traditional distributed systems. Existing research into exception handling for mobile agents is then summarised and problem areas are identified.

Chapter 4 provides a failure model for mobile agent systems. The failure model provides a classification of the failures that can occur in a mobile agent system. Specific attention is given towards the issues involved and assumptions necessary for crash failures of remote agent servers. Subsequently, a model for exception handling in mobile agent systems is then outlined and a conceptual exception handling scheme is proposed to protect mobile agents from agent server crash failures. The chapter concludes with a description of an IBM Aglets [Oshima98] implementation of the exception handling scheme.

Chapter 5 describes a case study application that is used to assist with the evaluation of the conceptual exception handling scheme outlined in chapter 4. The experiments that will be performed with the case study application, using an Ajanta [Tripathi02] and IBM Aglets [Oshima98] implementation of the exception handling scheme, are then described.

Chapter 6 evaluates the exception handling scheme outlined in chapter 4. The results of the case study experiments, outlined in chapter 5, are presented and evaluated. The exception handling scheme is then evaluated with respect to the exception handling model proposed for mobile agent systems in chapter 4. Finally, the features of the exception handling scheme are compared to existing systems that protect mobile agents from agent server crash failures.

Chapter 7 provides a summary of the research and some of the conclusions that can be drawn. The chapter also revisits the criteria for success listed in section 5. Finally, areas of future research are outlined.

Chapter 2 Mobile Agents

This chapter describes the broad context of the research, i.e. mobile agents in general. Later chapters focus on fault tolerance and fault tolerance for mobile agents. Mobile agents are defined for use in the thesis. A generic software architecture for mobile agent systems is then defined and some example application domains are outlined. Finally, the chapter concludes by discussing some of the problems with mobile agents.

1 What is a mobile agent?

This section aims to define the term *mobile agent* in the context of this thesis. There are many definitions for a mobile agent, dispersed across the distributed systems and artificial intelligence communities. This is more than likely attributed to the fact that the term *agent* itself is weakly defined, governed by many different classifications. Consequently, section 1.1 starts by highlighting the key properties of an agent. Section 1.2 then outlines the different types of agent within the research communities. Within the distributed systems research community mobile agents are often associated with mobile code systems. Significant confusion exists, distinguishing between mobile code systems and mobile agents. Consequently, section 1.3 defines the concept of a mobile code system and section 1.4 then distinguishes between mobile code and mobile agents for the context of the thesis.

1.1 What is an agent?

The agent community lacks a concrete definition for the term *agent*, a fact highlighted by Hyacinth Nwana [Nwana96] in her survey of software agent technologies:

> "*We have as much chance of agreeing on a consensus definition for the word agent as AI researchers have of arriving at one for artificial intelligence itself - nil! Recent postings to the software agents mailing list (agents@sunlabs.eng.Sun.COM) prove this*". [Nwana96]

This section defines the term *agent* by highlighting the key properties that distinguish agents from traditional software.

The term *agent* is often used as an umbrella term. Consequently, there have been many attempts to distil the key properties of an agent [Franklin97, Wooldridge97, Ndumu97]. Table 2-1 summarises properties frequently used to describe an agent.

Property	Description
Autonomy	The ability of the agent to act independently without human intervention.
Proactiveness	The agent performs actions in a goal oriented manner.
Reactive	Agents are situated in an environment, are able to perceive it and respond in a timely fashion to changes that occur in it.

Table 2-1 Agent properties

Autonomy is often cited as the significant property that distinguishes agents from traditional software. However, it is difficult to define. This is highlighted by Shoham:

> "*The sense of autonomy is not precise, but the term is taken to mean that the agents' activities do not require human guidance or intervention*". [Shoham93]

Wooldridge's perception of autonomy is that an agent's behaviour is denoted purely by its encapsulated state [Wooldridge97]. Huhns and Singh [Huhns97] extend this by introducing different degrees of autonomy:

- **Absolute**: The behaviour of the agent is completely unpredictable.
- **Social**: The agent is aware of communication partners but exhibits autonomy when independently entering commitments. Social autonomy is perceived as weaker, since some degree of autonomy is lost through co-ordination and communication.
- **Interface**: Autonomy is relative to internal design, provided that an application programming interface (API) is maintained. The API defines the behaviour of the agent but does not guarantee that it will continuously perform as requested.
- **Execution**: The degree of freedom the agent has while executing in its environment without external intervention.
- **Design**: The degree of autonomy assigned by agent designers. The greater the degree of autonomy the more heterogeneous the system is, since programmers contributing agents have to satisfy fewer constraints. Agent frameworks restrict design autonomy, e.g. some require that agents are constructed using a specific language.

The above definitions of autonomy highlight an agent's independence, i.e. the agent is capable of making its own decisions without external intervention. Conversely, traditional software does not usually possess this attribute since behaviour is imperatively declared.

Another property that distinguishes agents from traditional software is *proactive* or goal-oriented behaviour. An agent acts to satisfy a declarative set of goals. Consequently, the agent is expected to determine goal achievement possibly using reasoning or pre-compiled plans.

Therefore an agent must be aware of its goals. Conversely, a traditional program possesses goals, although they are implicit, i.e. hard coded.

So far the term *weak agency* has been defined to describe agents that are autonomous, proactive and reactive. Within the artificial intelligence community a stronger abstraction is considered, whereby the state of an agent consists of mental attitudes that the agent reasons with to decide subsequent actions. Attitudes explain human behaviour, e.g. "Simon turned down the thermostat because he *believed* the room was too warm." Wooldridge and Jennings [Wooldridge94] identify two mental attitudes:

- **Informational attitudes**: Belief and knowledge.
- **Pro-attitudes**: Desire, intention, obligation, commitment and choice.

Informational attitudes represent information the agent possesses about its environment and pro-attitudes direct or motivate the actions performed by the agent. Agents that possess strong agency are termed cognitive or *intentional systems*. The philosopher Dennett coined the term intentional systems to describe entities whose behaviour can be predicted by the method of attributing belief, desires and rational acumen [Dennett87].

Indeed an intentional system may be perceived as an abstraction tool that describes and predicts the behaviour of a complex entity. A popular model of an intentional system's state is the Belief-Desire-Intention (BDI) model proposed by Rao and Georgeff [Rao95]. Essentially, beliefs are facts the agent possesses about its environment. Desires are facts that the agent may possess in future states of the world, e.g. the ability to swim, and are consequently motivations. Intentions represent desires that the agent has selected to achieve. Intentions affect future decision making, i.e. an agent must not select intentions that conflict with those it currently holds.

Wooldridge and Fischer [Wooldridge94b] describe the advantages of the intentional abstraction compared to traditional software as:

- Non-technical and non-implementation dependent.
- Agents can model other agents. This is essential for certain types of collaboration.

However, selecting the beliefs and desires depends upon the designer's intuition of the modelled entity's role.

Figure 2-1 illustrates the perception of an agent adopted by the thesis. Agents are situated in an environment (e.g. the Internet) comprising users, hardware, agents, databases etc. Changes within the environment are actively monitored by the agent's sensors. The agent responds and acts upon changes through its effectors. Actions performed affect future sensing. Environmental changes may be triggered by some internal environment event or communication from a peer agent.

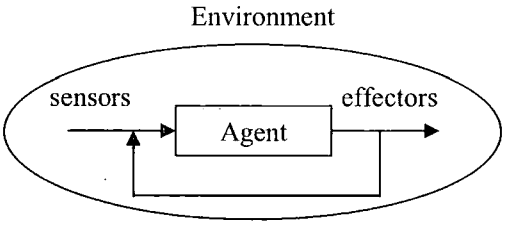


Figure 2-1 Reactive agent system

1.2 Types of agent

Hyacinth Nwana [Nwana96] provides a classification of agents. The classification of agents is summarised as follows.

Collaborative agents are characterised as autonomous, co-operative, static and coarse grained. Other traditional properties, such as learning and pro-activeness are common. The goal of a collaborative agent system is that the group of agents function beyond the capabilities of the individual members. Some of the motivations for a collaborative agent system are identified as:

- Overcoming resource limitation and system failure, the shortfalls of a single centralised system.
- Integration of legacy systems such as expert systems, decision support etc.
- Distributed information retrieval.

Interface agents are personal assistants that perform a task on behalf of a user. The agent observes the user performing a task, learns their activities and suggests alternative ways of performing them. Consequently, collaboration emerges between the agent and the user. Key characteristics are autonomy and learning abilities. Learning may involve:

- Observing and imitating the user.
- Receiving positive and negative feedback.
- Receiving explicit instructions from the user.
- Receiving advice from other agents.

Pattie Maes [Maes95] envisages interface agents providing a *proactive interface* as opposed to the traditional *direct manipulation* GUI. A direct manipulation interface passively waits for explicit instructions from the user. Conversely, a user may delegate tasks to a proactive interface agent that may perform complex tasks such as distributed information retrieval. In this sense, the user and interface engage in a co-operative process. Benefits include reducing the user workload for laborious tasks and adapting to user preferences and activities. An example is a web searching agent proposed in Lieberman [Lieberman95]. Traditionally, search engines are idle when the user browses. Similarly, the user is idle when a search is conducted. The proposed system uses agents to conduct a concurrent breadth search strategy based upon user browsing activity. For example, the user may frequently visit a site or reference a bookmarked page. If the agents are used within an Internet environment they are termed *Information/Internet* agents. Typically, these agents claim to address the problem of information overload and provide information management. Internet agents may be mobile, i.e. traverse the web gathering information and reporting the results achieved.

Mobile agents are capable of roaming a network, interacting with other agents at foreign hosts, gathering information and returning the results back to users [Nwana96]. In her survey paper [Maes95] Maes conveys the key attributes of mobile agents to be autonomy and co-operation.

Hybrid agents combine two or more agents into a singular agent so that the benefits of each agent type are maximised and the weaknesses are minimised. Hybrid agents are typically synonymous with hybrid agent architectures. For example, Muller *et al's* InteRRaP architecture [Muller96] combines deliberative and reactive agents.

A *heterogeneous agent system* comprises an integrated set of different types of agent, including hybrid agents. Consequently, agent programs can interoperate. A claimed advantage is that the cost of maintaining and rewriting legacy systems is reduced. A new domain, agent-based software engineering, facilitates interoperable agents. In particular, many standards have been proposed for heterogeneous agent systems. The key consideration is a base communication language. Genesereth and Ketchpel [Genesereth94] introduce the notion of an Agent Communication Language (ACL) that consists of the Knowledge Interchange Format (KIF), the Knowledge Query and Manipulation Language (KQML) and Ontolingua.

Nwana [Nwana96] classifies a reactive, hybrid and heterogeneous agent system as types of agent. Reactivity is defined in Table 2-1. In this case the types are architectural, providing the underlying agent technology.

Many more types of agent exist, the most notable being introduced in Franklin and Graesser [Franklin97] and Wooldridge and Jennings [Wooldridge95]. The following classifications are identified by Franklin and Graesser [Franklin97]:

- **Control / structure**: Deliberative or reactive.
- **Environment**: Database, file system, network and internet.
- **Language**: Interpreted or compiled.
- **Applications**: The application domains, e.g. electronic commerce.
- **Mobile**: Static or mobile.
- **Communication**: Communicative or non-communicative.
- **Adaptive**: Learning or non-learning.

Wooldridge and Jennings [Wooldridge95] envisage a broader classification according to the behaviour of the agent:

- **Gopher**: Perform simple tasks based on static rules and assumptions, e.g. reminder agents.
- **Service-performing**: Perform a well-defined high-level task as represented by the user, e.g. web searching.
- **Predictive/proactive**: Provide information or services to the user.

1.3 Mobile code systems

Fuggetta *et al.* describe a model for mobile code systems [Fuggetta98]. A mobile code system is a layered architecture comprising: hardware, a *core operating system*, *a network operating system*, a *computational environment* and *components*. The core operating system provides system services such as memory management, scheduling etc. The network operating system provides low level communication services such as the TCP/IP protocol etc. A computational environment provides applications with the ability to relocate components at different hosts. Relocation is understood as dynamically binding code and data to execution location. A component can be an *executing unit* or *resource*. An *executing unit* is synonymous to a thread comprising code and state (data space and execution state). Code is static and represents the behaviour of the executing unit. State consists of a *data space* and *execution state*. The data space represents the resources referenced by the executing unit. Resources may be shared by executing units, e.g. files, objects etc., and may be distributed at other computational environments. The execution state comprises thread data, variables and execution context, i.e. the stack and program counter etc.

Mobile code systems are distinguished from traditional distributed systems by the ability to move the code and state of an executing unit to a remote computational environment. There are two classes of mobility: *strong* and *weak*. A system possesses strong mobility when the entire

executing unit and execution state is transferred to a remote computational environment. Two mechanisms support strong mobility: *migration* and *remote cloning*. Migration mechanisms suspend the executing unit and transmit it to the destination computational environment, where it resumes execution. An executing unit migrates *proactively* or *reactively*. An executing unit that migrates proactively decides both the time and location of migration. An executing unit that migrates reactively does so in reaction to stimuli from a local or remote executing unit. Remote cloning mechanisms create a copy of an executing unit at a remote computational environment. Consequently, the original execution unit is static, since it remains at its current computational environment. Weak mobility mechanisms only support code relocation, i.e. code is transferred to another computational environment where it is either dynamically linked to a running executing unit or used to form a new executing unit. Note that execution state is not saved.

So far, migration of code and execution state has been described. However, an executing unit also references resources such as files, objects etc. that are owned by the current computational environment. An executing unit that moves to a new computational environment may still need to use resources at previous computational environments. However, resources are not always transferable. For example, an object may be shared between executing units or a file resource may be too large to move for performance reasons. The following techniques are suggested [Fuggetta98] for managing resources when an executing unit migrates:

- **Move**: If the resource is transferable, move it with the executing unit. An exception will be raised if other executing units attempt to reference the resource at the source computational environment.
- **Move with network reference**: If the resource is transferable, move it with the executing unit. Executing units at the source computational environment reference the resource at its new computational environment.
- **Network reference**: If the resource is not transferable, then the executing unit references the resource located at the fixed computational environment.
- **Copy**: Move a copy of the resource with the executing unit.
- **Locate compatible resource**: The executing unit locates a resource of the same type at the new computational environment.

1.4 Mobile code and mobile agents

Typical of evolving and immature research fields is the lack of widely accepted terms and methodologies. Mobile agents are no exception. In particular, there exists terminological and semantic confusion for distinguishing between *mobile code* and *mobile agents*. Sections 1.1 and 1.2 define and outline types of agent. Here it is seen that agent technology is often associated with the artificial intelligence community. However, mobile agents introduce an overlap between artificial intelligence and distributed systems, i.e. agent mobility implies mobile code

mechanisms. This section provides a brief history of mobile code and identifies the properties that distinguish mobile agents from mobile code.

Mobile code is by no means a new technology. Several mechanisms have been proposed for moving code between network nodes. The earliest technology is remote batch job submissions [Boggs73] and, surprisingly, the use of a page description language, postscript [Adobe85], to control printers. Nuttall [Nuttall94] provides a survey of *process* and *object migration systems*. Distributed operating systems employ process migration mechanisms allowing an operating system process to move from one machine environment to another and resume execution. In this case migration is transparent, i.e. the programmer is unaware of the process and has no control. A finer degree of mobility is provided by object migration mechanisms, allowing objects to move between address spaces. In this case, the programmer is able to specify what is migrated, ranging from atomic data to complex objects. In some cases the programmer may explicitly specify the location. Emerald [Levy88] is one such example.

In traditional distributed systems the elements required to perform a software service, i.e. code and resources, are all located at the same host. A client uses the service from a remote location by issuing method calls to a software component that performs the service. The software component is co-located with the code and resources necessary to perform the service. Conversely, mobile code systems allow the elements of a software service to relocate dynamically. Mobile code systems are therefore classified according to which elements are relocated. There are three classifications of code mobility [Fuggetta98, Picco01]: Remote Evaluation, Code on Demand and Mobile Agent. *Remote Evaluation* (REV) relocates the code to a remote host that holds the resources necessary for the computation. A *Code on Demand* (COD) system possesses the resources necessary for computation but downloads code dynamically from a remote host to perform the service. *Mobile agents* possess code and some of the resources necessary to perform the service. Migration to a remote host is driven by resource availability. In this case, the entire computational component migrates to the remote host where it resumes execution.

So far, it is has been established that a mobile agent is a classification of code mobility. Code mobility mechanisms are employed to move an entire component (mobile agent), i.e. code data and execution state, to a remote host. Furthermore, the mobile agent may move some of the resources to the remote host. Conversely, Remote Evaluation and Code on Demand mobile code systems only move code, i.e. resources at remote hosts remain static. This distinction is also highlighted by Luca Cardelli's definition of mobile code.

"*An architecture independent representation of program code (source text or byte codes) is shipped over the network and interpreted remotely. When code moves, the current state of the computation (if any) is lost, and connections that the computation had at the originating site vanish. State and connectivity must be re-established at the receiving site*". [Cardelli97]

Mobile code is therefore used to describe the ability to relocate code within a heterogeneous system. Upon arrival at a remote host, the code entity is interpreted and executed locally. State and resources accessed at the originating node are not preserved.

Many mobile agent definitions have been proposed within the artificial intelligence and distributed systems community. Furthermore, these are highlighted in the context of the underlying research project. Kotz *et al.* [Kotz99] define mobile agents as:

"*Programs that can migrate from host to host in a network, at times and places of their own choosing. The state of the program is saved, transported to the new host and restored, allowing the program to continue where it left off*". [Kotz99]

Luca Cardelli [Cardelli97] provides a contrast between mobile agents and mobile code:

"*Agents however, are meant to be completely self-contained. They do not communicate remotely with other agents; rather they move to some location and communicate locally when they get there*". [Cardelli97]

The above definitions summarise the autonomous nature of a mobile agent. A mobile agent possesses autonomy when it is an independent entity capable of deciding dynamically upon the time and location for migration. This implies that mobile agents must be aware of the resources available at the execution environment and respond appropriately to changes in availability.

Table 2-2 illustrates a comparison between mobile agents and mobile code. A mobile agent is an implementation of mobile code technology that is autonomous and capable of migrating its entire execution unit to a foreign host. Essentially, interactions are based on a computational component that communicates by relocating dynamically to another execution environment to access resources locally. This differs from mobile code technology, such as Code on Demand (COD) and Remote Evaluation (REV), whereby interactions are requests for remote execution of code. Formally, a mobile agent is understood to be a *self contained entity that is situated in an environment (e.g. the Internet), encapsulates state comprising code and data, can decide upon the time and location for migration and is resource aware.* Mobile agents are assumed to support weak mobility to compensate for the change of environment and consequent resource availability.

Property	Mobile Code	Mobile Agent
Mobility	Code	Code, data and possibly execution state.
Autonomous	No	Yes
Interaction	Request for remote execution of code.	Method invocation and migration to remote environment.
Resource aware	Yes	Yes

Table 2-2 Comparison between mobile agents and mobile code

2 Mobile agent architectures

Sections 1.1 and 1.2 introduced an *agent* as an umbrella term. The term *mobile agent* is used within the artificial intelligence and distributed systems communities. Mobile agents in the distributed systems community are communication-oriented, i.e. they provide code mobility, security and information passing. However, reasoning is performed at a low level using conditional statements. The artificial intelligence community focuses on agent knowledge representation and reasoning about the environment. For example, an intelligent mobile agent may be able to determine the fastest and most reliable route when it migrates to the next host. This thesis is concerned with mobile agent architectures for the distributed systems community. Consequently, mobile agents are believed to be reactive, responding to events such as migration requests and messages received from peer mobile agents. A mobile agent is an active object that consists of code, data and execution state. Each mobile agent executes within its own thread of control and, upon migration, the entire object graph is transferred.

The following structure is employed to describe the architecture of mobile agent systems in the distributed systems community. Section 2.1 starts by defining the term *software architecture*. Section 2.2 then outlines a generic mobile agent system and identifies the abstract elements, their functionality and interactions. There are two levels of interest with respect to the architecture of a mobile agent system. The first is the mobile agent. For example, how does a mobile agent organise its travel plan and communicate with resources (files and objects) at execution environments? These issues are described in Section 2.3. The second level of interest is the architecture of the execution environment. Sections 2.4 and 2.5 describe the core mechanisms of mobility provided by the execution environment, i.e. migration and intra mobile agent communication respectively. Finally, significant interest exists for interoperable mobile agent systems, i.e. mobile agents can migrate to different mobile agent systems and communicate with mobile agents from different vendors. Section 2.6 highlights standards and middleware for interoperability between mobile agent systems. Section 2.7 then concludes with a summary of the architectural features offered by existing mobile agent systems.

2.1 Software architecture definition

The architecture of a software system defines the system in terms of components and interactions among these components [Shaw95]. It can be considered as a high level abstraction for describing the structure of software, identifying entities, their functionality and interrelationships. Software architecture is certainly not a new field. Computing has witnessed network and hardware architectures. Software engineers may use architectures to provide users with multiple views to understand different aspects of the software structure. Perry *et al.* [Perry92] define a model for software architecture that consists of *elements*, *form* and *rationale*. Elements are classified as *processing, data* and *connecting*. Processing elements transform data elements that represent information. Connecting elements describe the interactions that integrate procedure and data elements, e.g. procedure calls, message passing and shared data. Form comprises *weighted properties* and *relationships*. Relationships specify the organisation of elements and interaction constraints. Properties are constraints on elements. A weighting grades the importance for a specific property/relationship or represents the degree of selecting from alternatives. Rationale relates to the motivation behind selecting the style of architecture and satisfying constraints such as functional and non-functional requirements.

In this thesis, architecture describes the design of a mobile agent system. The internal structure of a mobile agent system is characterised by identifying the abstract elements in terms of functionality and interactions. In particular, the architecture of the *agent server* is examined. The agent server provides an execution environment that is responsible for hosting software resources such as files and shared objects, managing mobile agent execution and transparently migrating the mobile agent to a remote host specified by the application developer. At another level the internal structure of a mobile agent is described, focusing on how the mobile agent interacts with its environment and organises migration to remote hosts. In the following section the elements that constitute a mobile agent system are described.

2.2 A generic mobile agent system architecture

Figure 2-2 illustrates an abstract architecture for a mobile agent system from the distributed systems community. Dashed entities are only provided by some mobile agent systems.

A mobile agent system consists of an *agent server* that runs within an interpreter and is capable of hosting *mobile agents* implemented in the interpreted language. Consequently, running the agent server within an interpreter provides operating system interoperability and the mobile agent can migrate between hosts independent of the operating system. This assumes that the same agent server and interpreter exist at each host. An *agent server* is a server process that runs at hosts willing to accept mobile agents.

The agent server has the following responsibilities:

- Pack and unpack mobile agent code, data and execution state.
- Transport the mobile agent to destination host.
- Authenticate the identity of a mobile agent's owner.
- Enforce resource access limits to visiting mobile agents.
- Provide a communications infrastructure for local mobile agent interaction.

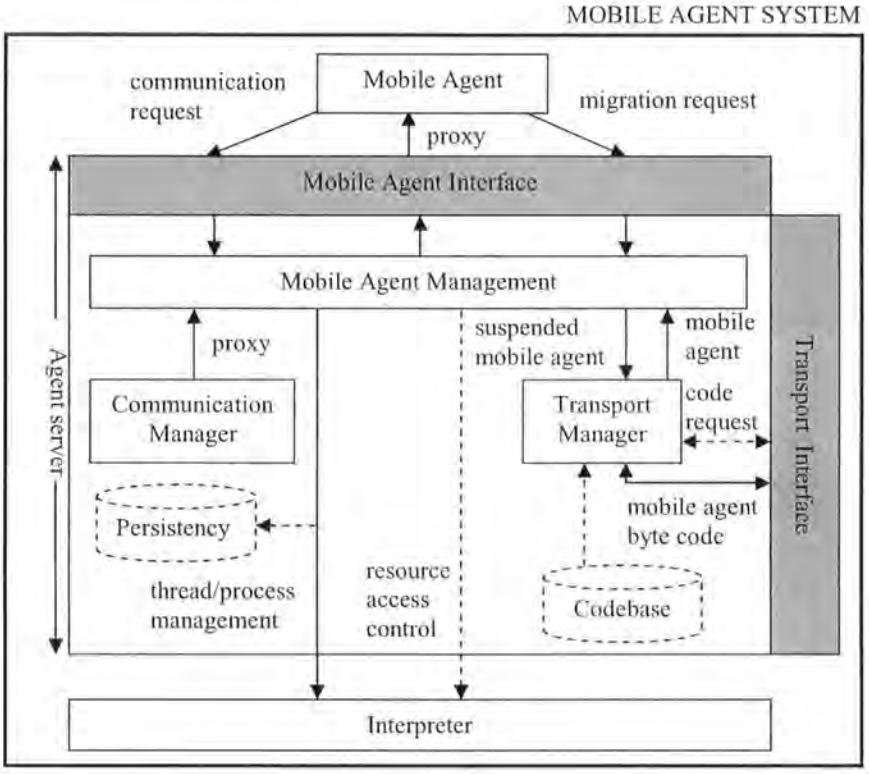


Figure 2-2 Mobile agent system architecture

Figure 2-2 illustrates that the agent server provides a *mobile agent interface* and a *transport interface*. The mobile agent interface allows mobile agents to interact with the agent server for migration and communication requests. The transport interface is provided for agent servers to send and receive mobile agents over a secure communication channel. The transport interface may also provide access to a code repository if code is unavailable at the remote agent server. Typically, the transport interface uses TCP, since it is a connection oriented protocol that offers reliability. The core components of the agent server are:

- **Agent management:** Responsible for mobile agent management (initiation, suspension, resumption and termination) and routing requests to the transport and communication manager respectively. The agent management component is also responsible for creating a unit of execution for the mobile agent within the interpreter and suspending the mobile agent after a migration request.

- **Transport manager**: Responsible for the capture and restoration of mobile agent state, i.e. data variables, code, and execution state (if strong mobility semantics are assumed). A *code base* is provided by some agent servers to download code for the mobile agent if it is not available at remote agent servers. There are different strategies to download mobile agent code. For example, some agent servers ship the entire code with the mobile agent. Others request code from the code base, as and when it is needed. Section 2.4 describes the notion of mobile agent state and code download strategies.
- **Communication manager**: Responsible for intra mobile agent communication, i.e. mobile agent communication with local mobile agents. Communication between mobile agents is possible using method calls and shared memory. Most agent servers only provide some of these communication mechanisms. For example, Tracy [Braun01] provides shared memory and message passing. Section 2.5 discusses mobile agent communication mechanisms.

Unfortunately, a variety of agent servers are implemented using different interpreted languages. Furthermore, the mobile agent and transport interfaces differ between agent servers. Consequently, migration between agent servers is mostly homogeneous, i.e. a mobile agent can only migrate to hosts that offer the same agent server and interpreter. However, recently there has been significant interest regarding interoperability between mobile agent systems. Section 2.6 describes the current approaches for interoperable mobile agent systems.

2.3 Mobile agent architecture

A mobile agent is typically represented by an object that inherits from an abstract class, unique to each mobile agent system. The abstract class provides a method, e.g. *run()*, to represent the main execution thread of the mobile agent. Instructions are provided by the agent server to move to another host, e.g. *go(agent server address)*. An object-oriented representation of a mobile agent is adopted in this thesis due to maintainability and the wide choice of available mobile agent systems. However, there are other representations of a mobile agent that are not object-oriented. For example D'Agents [Gray02] and TACOMA [Johansen02] allow mobile agents to be written using the interpreted Tcl procedural scripting language.

Information available to the mobile agent includes: the address of the *home agent server*, i.e. the agent server that created the mobile agent, its identification, an interface to the current agent server environment and an *itinerary* of agent servers to visit. Each agent server provides an object interface that a visiting mobile agent can use to access:

- The name or address of the current agent server.
- A registry of mobile agents currently running at the agent server.
- A registry of resources available for consumption at the agent server.

When the mobile agent migrates to a new agent server a reference to the agent server environment is updated. Using the resource registry, a mobile agent can communicate with co-located agents to access and publish resources at the agent server. A resource may be a file, database or software object. For example, an agent server may represent a supplier that provides an interface to a database that represents its product catalogue. Mobile agents may interact with the resource to search the catalogue for specific items of interest. Directory services for traditional distributed systems are employed in most mobile agent systems. Other mobile agent systems, such as IBM Aglets [Oshima98], require the programmer to employ static mobile agents to represent services. In this case, a list of mobile agents running at the agent server is accessible to the mobile agent. Each directory entry corresponds to an object that is referenced by an immutable name, e.g. a URL. Clients retrieve an object from the directory to access information or perform an action. A proxy is returned that forwards the request to the object. Java mobile agent systems employ Java directory services, e.g. the RMI naming registry or JNDI. Mobile agent systems compliant with the MASIF standard, e.g. Grasshopper [Baeumer03], employ the CORBA naming service.

A mobile agent uses an itinerary data structure as an organised representation of the agent servers to visit during its trip. An itinerary can represent an ordered or dynamic set of agent servers. Each entry in the itinerary represents the address of an agent server. The simplest representation of an itinerary is an ordered list of agent server addresses usually with an index to represent the current agent server. In addition to storing the agent server address some itineraries also log the method that will be invoked upon the mobile agent's arrival. Consequently, it is possible to invoke a different method depending upon the current location of the mobile agent. Itineraries that employ the same method at each agent server are classified as fixed entry. Most itineraries do not store the success of each visit for an agent server. For example, a mobile agent may visit an agent server and encounter a software exception, e.g. the agent server may deny access to its resources. The Ajanta [Tripathi02] mobile agent system allows the success of each agent server visit to be logged. Other mobile agent systems [Braun01, Gray02, Oshima98, Peine02] leave this responsibility to the programmer. Furthermore, Ajanta [Tripathi02] also allows the programmer to define criteria for choosing the agent server to visit using its *Select* itinerary pattern. For example, criteria may include the availability of the agent server or the state of the mobile agent. Alternatively, the *Set* itinerary selects the next unvisited agent server at random.

There are various design patterns for an itinerary [Oshima98, Tripathi02]. IBM Aglets [Oshima98] provide a master slave pattern that allows a stationary master agent to spawn one or more slaves that migrate and execute in parallel. When a slave's itinerary is complete, results are returned to the master. Another example is the meeting pattern that allows a group of mobile

agents to interact locally at a single host and exist independently of the home agent server. The address of the agent server where the meeting occurs is pre-arranged. Each agent has a meeting object that stores its location and id. When a mobile agent arrives at the agreed agent server, the meeting object informs a meeting manager at the agent server. The meeting manager notifies all agents that have arrived in the group of the new arrival.

2.4 Mobile agent migration

Migration describes the process of packaging the code, data and execution state (if strong mobility semantics are adopted) for a mobile agent into a portable representation that is transported on a TCP network connection to the destination agent server. The migration process involves sending and restoring a portable representation of a mobile agent at a remote agent server. Before outlining the process for migration it is necessary to understand how the code and data of a mobile agent can be represented, the strategies for code mobility and resource maintenance.

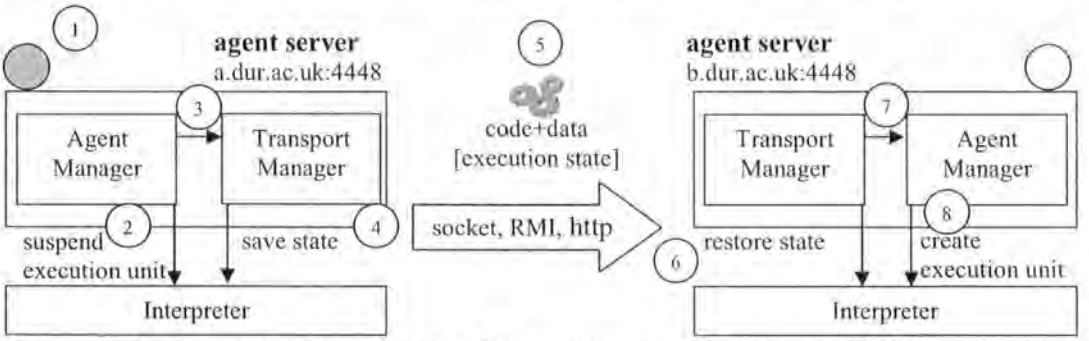
There are many Java mobile agent systems, since persistency and code mobility are provided as standard language features. Firstly, object persistency is provided by the Java serialisation API. In Java, *serialisation* denotes the activity for storing the state of an object and its set of objects into a serial form, i.e. bytes. *Deserialisation* is the activity of restoring the state of an object and its associated set of objects from its serialised form. Details concerning the class of each object, such as the name and version number, are included in addition to the types, names and values of instance variables. This meta-data is used to restore the state of the object. However, an object cannot be instantiated without an associated class file which represents the behaviour, i.e. methods. Java allows the programmer to customise class loading. Custom class loaders are used when the default Java class loader cannot locate a class in either the local cache or the directories specified in the CLASSPATH system variable. Consequently, the programmer can dynamically load the classes from a remote location over the network. Despite the popularity of Java for mobile agent systems, alternative implementations do exist in other languages [Gray02, Peine02]. These provide a customised representation of code and data. For example, Lingau *et al.* [Lingau95] embed the code and data into a Multipurpose Internet Mail Extension (MIME) message. Others use XML [Emmerich00] or provide a custom representation of transforming the code and data into bytes [Gray02, Peine02] for shipment to the destination agent server.

So far, it has been established that there are numerous strategies for transferring data and code. However, there are also different strategies for code mobility, i.e. *pull*, *push* or *push per unit*. A *pull* code mobility strategy sends only the mobile agent's class to the destination agent server. When the destination agent server unpacks the data and code, it requests classes from the

mobile agent's home agent server, i.e. the agent server that created the mobile agent. Consequently, code is downloaded on demand from the home agent server, as and when it is needed. This approach has the disadvantage that the autonomy of the mobile agent is reduced, i.e. if the home agent server fails then the mobile agent can no longer execute. Conversely, the *push* mobile code strategy ships all classes to the destination mobile agent server. This is popular with non Java implementations and offers the advantage that the mobile agent is independent of the home agent server, i.e. the mobile agent can execute even if the home agent server is disconnected. However, shipping the entire classes to the destination agent server may represent inefficient use of network bandwidth, i.e. some classes may not be needed at the destination agent server. Alternatively, there are some mobile agent systems, e.g. Tracy [Braun01] and Sumatra [Acharya97], which employ a *push per unit* strategy to enable the programmer to specify the classes that migrate with the mobile agent.

The majority of mobile agent systems ship the entire object graph occupied by the mobile agent. This means that the state of all objects referenced by the mobile agent is saved. Consequently, the entire mobile unit migrates to a new location. In these systems it is the responsibility of the programmer to maintain references to stationary objects that exist at the remote agent server. For example, a stationary object may represent a resource that the mobile agent wishes to use. Some mobile agent systems allow the application developer to specify co-location semantics with respect to static resources at remote sites and peer units of mobility [Holder99, Picco98]. For example, Fargo [Holder99] allows the programmer to select related units of mobility to migrate. In Fargo [Holder99] a unit of mobility is represented by a group of objects termed a *complet*. The programmer can choose to move referenced complets, move a duplicate of a referenced complet or form a new reference to a complet of a similar type at the remote destination. Unfortunately, mobile agent systems that provide this functionality are rare.

Figure 2-3 illustrates the process of migration between two agent servers. When a mobile agent issues a migrate request (1) the unit of execution for the mobile agent is suspended by the agent manager (2) and forwarded to the transport manager (3). In Java mobile agent systems the



Agent servers are assumed to run the same interpreter and agent server platform.

Figure 2-3 The mobile agent migration process

unit of execution corresponds to a thread that executes within an agent server process. Others, e.g. [Gray02], run the mobile agent in an operating system process. The state of the mobile agent is then packaged (4) and transferred (5) to the destination agent server by the transport manager. Mobile agent systems differ according to the transport protocol used. For example, each agent server may provide an RPC interface with a signature to receive a portable representation of the mobile agent. Others may provide TCP socket or HTTP interface. Upon reception of state, the transport manager at the destination agent server unpacks the state (6) and forwards it to the agent manager (7) that creates a new unit of execution (8).

The process of packing and extracting the state of the mobile agent depends upon the mobility semantics and the interpreter adopted by the mobile agent system. For example, if the interpreter allows saving the execution state, e.g. D'Agents [Gray02], then the agent system developer can use the state extraction routines of the interpreter to provide strong mobility semantics. Alternatively, if the interpreter does not provide execution state restoration, e.g. Sun's Java Virtual Machine (JVM), then the agent system developer has more work to provide strong mobility semantics. Firstly, the interpreter could be modified to allow saving the execution state of the mobile agent thread. This approach, adopted by NOMADS [Suri00] and Sumatra [Acharya97], has the disadvantage that hosts must install a new interpreter. An alternative is to transform the bytecodes produced when the mobile agent is compiled. For example, Sakamoto *et al.* [Sakamoto00] propose an algorithm that transforms the bytecodes for each method by adding instructions to save and restore the execution state. This is achieved by modifying the method signature to include a data structure that stores the execution state. A method that contains a migration instruction is modified to throw an exception. This triggers saving the execution state of the method. The exception is then thrown to trigger state saving operations further up the calling hierarchy. Upon resumption, instructions are inserted to restore the execution state using the data structure passed to the modified method signature. An alternative method is to transform the source code to produce a class file that provides strong mobility [Sekiguchi99]. This works on the same principle, i.e. exceptions are used to trigger the state saving routine up the call hierarchy. However, a compiler must insert instructions before runtime and the approach requires access to the source code for the mobile agent.

2.5 Mobile agent communication

Communication between client and server software components in a conventional distributed system uses Remote Procedure Call (RPC), whereby a client requests the invocation of a remote method offered by a server. The server performs the request and replies with the result. Both the client and the server exist at fixed nodes and must be aware of the identities of communication partners. Consequently, communication is synchronous, i.e. the client blocks until the server sends an acknowledgement. The mobile agent research community is

speculative of using remote procedure calls for mobile agent interaction. Firstly, the remote procedure call communication strategy assumes that communication parties share a common name space and are aware of each other's identity. This can sacrifice the degree of a mobile agent's autonomy [Hermann00, Cabri02]. For example, if a mobile agent executes a procedure under foreign control, i.e. it is fulfilling a remote request, the agent may not migrate until the procedure has been executed. Equally, it could be the case that the mobile agent migrates without fulfilling the request. Secondly, communication relies upon the stability of the network. This is a problem when the location of the mobile agent is dynamic. A complex communications infrastructure is required for mobile agents to provide message forwarding and track the dynamic location of named mobile agents.

One solution is to reduce the complexity of the communications infrastructure by localising the mobile agent interactions and wherever possible, enforce uncoupled interactions [Cabri02]. Meeting points and tuple spaces are examples. Mobile agents can establish a meeting point at a local agent server where agents can join and synchronise for interaction. Establishing a meeting point at a local agent server reduces the problems of a stable network, since all communication occurs locally. However, all participants must be present at the same agent server, thus sacrificing a degree of autonomy. Furthermore, the meeting point represents a single point of failure. Tuple spaces provide *associative* communication and co-ordination. A tuple space communication model consists of a shared data space (*tuple space*), information elements that exist on the data space (*tuples*) and a small set of operations to add, remove and access tuples on the space. Communication is *associative*, i.e. information is accessed by content, possibly using pattern matching. Interactions are characterised as indirect and uncoupled through the creation and manipulation of tuples on the shared tuple space. Specifically, *temporal* and *spatial* decoupling is enabled [Murphy01]. Temporal decoupling means that communication parties do not need to synchronise for communication, i.e. the sender and the receiver do not have to exist at the same time. Spatial decoupling means that the communication parties do not have to be aware of the identities of their peers to communicate. Both properties preserve the autonomy of mobile agents. This means that the mobile agent does not have to lookup the name and location of a communication peer and then synchronise communication activity at a specific location. Furthermore, communication is localised to the tuple space. Linda [Gelernter85] was the earliest tuple space implementation that provided a single tuple space with no transactional support or authentication of tuple accesses. Later implementations, e.g. TSpaces [Lehman99] and JavaSpaces [JavaSpaces03], provide persistency and an event notification system that notifies registered communication parties when the tuple space is modified. In these implementations, a central tuple space provides remote access to multiple communication parties.

MARS [Cabri02, Omicini01] and TUCSON [Omicini01] provide *programmable tuple spaces* that are independent of Java mobile agent systems. An independent programmable tuple space exists at each agent server to allow reactions to be programmed for the addition, removal and access of tuples on the space. A reaction is represented as a code fragment, i.e. a Java object in MARS [Cabri00, Cabri02] and a first order logic term in TUCSON [Omicini01]. The code fragment is executed in response to a specific tuple operation. MARS [Cabri00, Cabri02] allows reactions to be installed by the agent server or application mobile agents. For example, the agent server may install reactions to provide environment specific co-ordination such as security. Alternatively, an application mobile agent may install a reaction onto the tuple space to exchange application specific knowledge. Application based reactions must be confined to the context of the application and are enforced by associating an identifier with mobile agents in the application. MARS [Cabri00, Cabri02] and TUCSON [Omicini01] provide a single static tuple space at each agent server. The tuple space is independent of tuple spaces at remote agent servers. Alternatively, LIME [Murphy01] permanently associates one or more named programmable tuple spaces with a mobile agent. Consequently, the tuple spaces migrate with the mobile agent. When a mobile agent arrives at a host the tuple spaces are automatically merged with the tuple spaces of co-located mobile agents. Consequently, mobile agents that are co-located at an agent server may share tuples for the duration of their stay. Furthermore, a mobile agent may declare private tuple spaces. When a mobile agent migrates to another host, the tuple spaces migrate with it. Consequently, upon migration, the tuples owned by the mobile agent are no longer visible to co-located mobile agents. Tuple spaces have potential for mobile agent communities that interact via accessing and manipulating shared data stored at one or more agent servers. LIME [Murphy01] allows a mobile agent to migrate a tuple space that represents the application context. However, the size of the tuple space must be controlled to preserve the potential bandwidth savings offered by mobile agent technology.

2.6 Mobile agent interoperability

A significant problem that hinders the widespread adoption of mobile agent technology is the lack of interoperability between mobile agent systems. Two mobile agent systems are interoperable if a mobile agent of one system can migrate to the second system, the agent can interact and communicate with other agents (local or even remote agents), and the agent can leave this system and resume its execution on the next interoperable system [Pinsdorf02]. The following issues contribute to the lack of interoperability between mobile agent systems:

- **Language**: Agent servers are typically implemented using an interpreted language, e.g. Java and Tcl. If hosts run the same agent server platform then interoperability between computer architectures is gained. For example, mobile agents are able to migrate between Microsoft Windows and Linux machines. However, migration between agent servers implemented using different languages is homogenous.

- **Mobile agent system interface**: Agent servers provide an interface for visiting mobile agents to access resources and platform services such as agent management, tracking etc. Different agent servers provide different interfaces for visiting mobile agents. Furthermore, most mobile agent systems enforce the application programmer to inherit from an abstract class. Unfortunately, even if agent servers are implemented using the same programming language, the abstract class is not universal across all mobile agent systems.
- **Migration semantics**: Most agent server platforms provide either strong or weak mobility. Sun's Java Virtual Machine does not provide execution state capture. Consequently, Java mobile agent systems that require strong mobility provide a modified Java Virtual Machine or use pre-processing. Consequently, migration between agent server platforms that provide different migration semantics is difficult.
- **Heavyweight architectures**: Most mobile agent systems are single monolithic heavyweight systems attempting to provide a common denominator of features [Picco98]. These features include communication (message transport and delivery), mobility (agent transport and encoding), security (agent authentication and state appraisal) and general (agent creation and lifecycle) [Pinsdorf02]. The same features are often implemented different ways across mobile agent systems. For example, most Java mobile agent systems provide some means of mobile agent communication using RMI. Others, such as D'Agents [Gray02], provide their own implementation. The μCode [Picco98] open source project focuses on this problem to provide a lightweight mobile agent system. This allows developers to select different implementations for the same feature.

Existing approaches for interoperable mobile agent systems either enforce standard interfaces or employ agent factories that convert a mobile agent into a mobile agent for the target environment. Each approach focuses on different levels of interoperability. For example, some provide interoperability between mobile agent systems implemented using the same interpreted language. Furthermore, only some aspects of interoperability may be enforced, e.g. security may not be addressed. This section describes the existing approaches for interoperability between mobile agent systems.

2.6.1 Interoperability standards

There are two interoperability standards for mobile agents, MASIF [Milojivcic98] and FIPA [Fipa04]. MASIF [Milojivcic98] incorporates interoperability using the CORBA framework without the need to modify the agent platform. Two CORBA IDL interfaces are provided, implementations of which can be published to a CORBA naming service. The *MAFAgentSystem*

interface must be implemented by the agent server to provide interoperability for agent transfer and management. Standard methods are provided concerning the mobile agent lifecycle (create, terminate, suspend and resume) The *MAFFinder* interface, accessed via the CORBA naming service or *MAFAgentSystem* interface, provides a registry that can be used to maintain a database of mobile agents, agent servers and mobile agent systems. Methods are defined to register, deregister and locate these entities. A mobile agent can also use the *MAFFinder* interface to locate mobile agent systems that match its requirements specified in an *AgentProfile* object. Requirements include language, serialisation mechanism, the type of agent server and version number. MASIF [Milojivcic98] uses these interfaces to standardise mobile agent and agent server names, agent system types and location syntax. It should be noted that there are some aspects that are not standardised. These include agent encoding and security. The only mechanism for agent transfer that is standardised is the interface for agent transfer employed by the mobile agent system. Consequently, the MASIF [Milojivcic98] standard assumes agreement between agent servers concerning the encoding of the mobile agent for transfer. Furthermore, it is clearly stated that the standard is only concerned with interoperability between mobility agent systems written in the same language and expected to go through revisions [Milojivcic98].

FIPA [Fipa04] is a standard from the intelligent agent community that focuses significantly on interoperable agent communication between heterogeneous FIPA compliant agents using a standard Agent Communication Language (ACL). The specification for the FIPA standard [Fipa04] outlines a framework for agents including an Agent Management System (AMS), Directory Facilitator (DF) and Agent Communication Channel (ACC) that operates over CORBA IIOP. The Agent Management System and Directory Facilitator offer functionality similar to the *MAFAgentSystem* and *MAFFinder* interfaces of the OMG MASIF [Milojivcic98] standard. Communication between agents is achieved using a message forwarding service between agents.

Grimstrup *et al.* [Grimstrup02] propose the GMAS interoperability standard to serve as an interface between the native mobile agent system and foreign mobile agents from different platforms. The interoperability standard translates a foreign mobile agent system API into the API for the native mobile agent system. Consequently, to achieve interoperability each mobile agent system must provide a translator between its own API and the GMAS interoperability API. This is achieved by implementing the *Foreign2GMAS* interface. To host foreign mobile agents, the native mobile agent system must implement the *GMAS2Native* interface that converts GMAS API calls into API calls for the native mobile agent system. Each mobile agent system has a *gateway* and *launcher* software component that is responsible for dispatching and receiving mobile agents respectively. Migration between GMAS compliant mobile agent systems is achieved by transporting the mobile agent state and meta-data between the gateway

at the origin and the launcher component at the target mobile agent system. However, it is assumed that a common communication protocol, e.g. CoABS [Coabs04], exists between mobile agent systems. When a mobile agent migrates to a different GMAS compliant mobile agent system, its Foreign2GMAS adaptor is dynamically downloaded from a specified remote code base. This then communicates with the GMAS2Native adaptor for the target mobile agent system.

2.6.2 Mobile agent factories

Brazier *et al.* [Brazier02] migrates an architectural description, or blueprint, of mobile agent functionality with the application state. An agent factory at the target mobile agent system uses the blueprint to generate a compatible mobile agent. The mobile agent functionality is specified at two levels. The conceptual level describes the components of the mobile agent including component interfaces and interactions. The detailed level includes code and definitions, e.g. interfaces. For example, there may be implementations in Python, C and Java for a single component at the conceptual level. Libraries of descriptions may include design patterns, knowledge based models or agent wrappers that provide cross platform interfaces. The agent itself is responsible for storing and restoring state using platform independent measures, e.g. XML. The agent factory is responsible for sending the agent state and blueprint to the target host. It is assumed that the target host has access to an agent factory capable of producing a mobile agent for the target platform. Migrating a specification of the mobile agent functionality offers the advantage of language interoperability. However, so far, this is only applicable for mobile agent systems that support weak mobility.

Design techniques exist for interoperability between Java mobile agent systems. A design based upon the adaptor pattern is presented in [Misikangas00]. This design separates platform specific functionality from the application. A mobile agent is separated into two classes, a head and body. The *head* is platform independent and represents the application mobile agent. The *body* represents the platform specific operations of the mobile agent with methods defined for migration, message passing and service location. Interoperability is achieved by moving the head to a target mobile agent system and binding it to the platform specific body implementation. A migration service (Monads Agent Gateway) is provided that opens a socket to the destination and transfers the head. The receiving migration service (Monads Agent Gateway) then binds an instance of its platform specific body to the received head. The migration service is only used when the mobile agent needs to migrate to a different mobile agent system and exists at hosts that provide the monads service API [Campadello00]. Interoperable communication is provided by using text based messaging. A method is provided in the body (*receiveMessage(String)*). If the head is present, the message is delivered. Otherwise, the body can either store the message until the head returns or request the migration

service to deliver the message to the head. However, no details are provided regarding how the gateway determines the current location of the mobile agent.

An agent factory has also been adopted by [Pinsdorf02] in the SeMoA mobile agent system [Roth01]. At the time of writing Jade [Bellifemine99] and Tracy [Braun01] mobile agents can migrate and execute at any SeMoA [Roth01] agent server. When a mobile agent is deserialised, a lifecycle registry forwards it to registered factories and waits for a signal from a factory that is willing to handle the mobile agent's class. The factory then generates an instance that handles the mobile agent's lifecycle. The instance acts as a wrapper capable of translating between native lifecycle and foreign mobile agent system lifecycle. Consequently, all necessary components are instantiated to make the mobile agent believe that it is running on its native mobile agent system. However, access to source code is assumed for analysis of the mobile agent architecture and lifecycle.

2.7 Mobile agent architecture summary

This section concludes by summarising the architectural features available for existing mobile agent systems. There are some mobile agent systems such as Concordia [Wong97] that are no longer available. For this reason these have not been included in the survey, since they do not represent the latest available mobile agent systems.

Table 2-3 characterises mobile agent systems according to the following properties:

- **Language**: The implementation language adopted by the mobile agent system for application developers to program a mobile agent.
- **Code transfer**: The strategy used to transfer mobile agent code. Section 2.4 introduced the push, pull and push per unit code download strategies.
- **Mobility semantics**: Is strong or weak mobility used for migration?
- **Communication mechanism**: The means of communication available to mobile agents. Section 2.5 introduced remote procedure calls, meeting points and tuple spaces as mechanisms for communication between mobile agents.
- **Interoperability**: Sections 2.6.1 and 2.6.2 described interoperability mechanisms for mobile agent systems. Interoperable mobile agent systems may implement a standard [Milojivcic98, Fipa04, Grimstrup02]. Alternatively, agent factories may be used to translate a mobile agent into a mobile agent for the target mobile agent system.

	Language	Code transfer	Mobility semantics	Communication semantics	Interoperable
IBM Aglets [Oshima98]	Java	push / pull (code server)	weak	remote and meeting point	partial MASIF
Ajanta [Tripathi02]	Java	pull from code server	weak	remote	none
Ara [Peine02]	Java, C++, Tcl	push	strong	remote and tuple space	none
DAgents [Gray02]	Java, Tcl, Scheme	push	strong	remote	none
Fargo [Holder99]	Java	push	weak	remote	none
Grasshopper [Baeumer03]	Java	pull	weak	remote	MASIF/FIPA
JSEAL2 [Binder01]	Java	push	weak	remote	none
Jumping beans [Beans04]	Java	push	weak	remote	none
Nomads [Suri00]	Java	push	strong	remote	none
SeMoA [Pinsdorf02, Roth01]	Java	push / pull (code server)	weak	remote events	Tracy and Jade mobile agent systems
Sumatra [Acharya97]	Java	push	strong	remote	none
Tracy [Braun01]	Java	push all and pull	weak	mailbox and tuple space	none

Table 2-3 Summary of mobile agent systems

Table 2-3 highlights the popularity of Java as the implementation language for mobile agent systems. Subsequently, weak mobility predominates over strong mobility, since without modification the Java Virtual Machine cannot serialise the execution state of threads. The default mobility strategy uses a pull strategy. However, some mobile agent systems provide a hybrid strategy to allow the programmer to control the strategy of mobility, e.g. Tracy [Braun01], SeMoA [Roth01] and IBM Aglets [Oshima98]. Interoperability between mobile agent systems is in the early stages. Most of the available mobile agent systems are either compliant with an interoperability standard [Milojivcic98, Fipa04] or provide interoperability at

the language level, i.e. mobile agents can be implemented using different languages. However, these systems still limit migration to hosts that run the same agent server platform. There is evidence of limited interoperability between some mobile agent systems, e.g. some developers have collaborated [Pinsdorf02] to provide interoperability only between the Jade [Bellifemine99], Tracy [Braun01] and SeMoA [Roth01] mobile agent systems. Section 2.6 highlighted that the trend is moving away from standards such as MASIF [Milojivcic98], toward middleware that uses adaptors to provide interoperability.

3 Mobile agent applications

A significant problem with mobile agents is the lack of a "killer application" [Milojivcic99]. Small case study applications have been used within the research community for proof of concept. However, there is little evidence of commercial applications that solely use migration. Consequently, the trend appears to be to focus not on finding applications that purely use mobility but rather on examining the conditions and scenarios where mobility is a useful tool in applications [Kotz02].

Mobile agents have been proposed for routing protocols in mobile ad hoc networks. A routing protocol directs traffic from a source to a destination node to maximise the network performance and minimise the costs. An ad hoc network is defined as a multi-hop network that consists of mobile hosts that communicate without the support of a wired backbone, HA/FA or Base Station [Wang01]. Multi-hop communication occurs when the mobile hosts are not in direct radio range and are routed through one or more intermediate mobile hosts [Liu02]. The network is characterised as ad hoc since the topology or structure of the network frequently changes. For example, a mobile host may move out of range from its neighbours to a new location and form new neighbours. Some conventional routing protocols such as Destination Sequenced Distance Vector [Perkins94] rely upon knowledge of the network topology *a priori*. These algorithms are unsuitable for use in a mobile ad hoc network environment, since topology information would frequently have to be distributed over the network due to the dynamic nature of the network [Marwaha02] and [RoyChoudhury00]. It is expected that this would seriously limit the network bandwidth for actual communication. Routing schemes such as Dynamic Source Routing (DSR) delay the transmission of data until the route is discovered, thus being unsuitable for real time systems. Furthermore, it would be difficult for mobile hosts to perform the routing algorithm, since they may not possess sufficient battery power for algorithm execution [Mingas03].

Mobile agents have been proposed to traverse the topology of the network and provide full connectivity information. The general strategy is that a population of mobile agents are frequently dispatched to a randomly selected destination in the network. Each mobile host owns

a routing table that mobile agents query to determine the next mobile host to visit in the path towards the destination. Mobile agents maintain a history of intermediate nodes visited on the path to the destination. Each entry in the history may be associated with a trip time denoting the distance between the launch host and intermediate hosts. Mobile agents update the routing table of mobile hosts with the best routes for other nodes in the network. Consequently, the mobile agents are active and possess intelligence for selecting the best route to each node visited on the path to a destination mobile host. However, there are problems with the mobile agent approach [Marwaha02]. A host sending packets to a destination for which it doesn't have an up-to-date route must wait for a mobile agent to provide a route. Furthermore, if a route breaks, the source may still keep sending data packets unaware of the link failure. A hybrid approach in [Marwaha02] combines the AODV [Perkins99] routing protocol with mobile agents to overcome this. AODV is used for local connection maintenance. Mobile agents enhance the shortfalls of AODV routing protocols by increasing connectivity and decreasing the end-to-end delay and route discovery latency.

In [Minar99] mobile agents co-operate to learn about the connectivity of the network. A mobile agent learns about all the edges for the node where it is located and stores them as knowledge. Next, the mobile agent learns node edges from peer mobile agents co-located at the current node. Finally, the agent selects a node to migrate to. Selection of the node is done conscientiously at runtime, by choosing a node that has been least visited or never before visited. Super conscientious agents also visited rarely visited nodes, but the decision to move is based on facts assembled independently and by peers. A simulation measured the time taken for all agents to learn the connectivity of the network. Obviously, conscientious agents perform better than using a random selection strategy, since the nodes acquire more information from each other. For small populations of agents it was found that the super conscientious selection strategy performed best. However, as the node population grew, super conscientious agents tended to be clustered, thus duplicating efforts based on shared information.

Mobile agents are useful for information retrieval applications, whereby a mobile agent visits one or more remote hosts to query and filter data. The hypothesis is that network bandwidth is utilised more efficiently by co-locating a mobile agent to one or more remote agent servers to perform computations on data locally. The success of the hypothesis depends upon the nature of the application [Picco01]. For example, if the mobile agent accumulates data at remote hosts, there is the danger that the size of the mobile agent outweighs the performance benefit of issuing separate RPC calls to the servers. Furthermore, employing mobile agent technology for message passing, e.g. querying a remote database, demonstrates poor performance compared to using traditional RPC calls. This is likely attributed to the overhead of migrating the code and the query on route to the remote host, in addition to the query results on

the return to the sending host. However, the hypothesis may be useful when the mobile agent is used to compress or filter the data at remote hosts. In [Picco98b] this approach was used in a network management application where a mobile agent migrated to remote hosts to determine the busiest network interface. It was found that this produced a saving of 30% on the utilisation of the network compared to using a centralised network management station that polled devices. Mobile agents have been used to implement distributed indexing [Grey00] to overcome the performance bottleneck given by centralised indexes. Each web site maintains its own index. A community of mobile agents wanders the web visiting remote sites seeking information of interest on the behalf of the user. Three types of agents are introduced: ferrets, publicists and gurus. Essentially, agents traverse the Internet looking for indexes that meet the needs of the user. Hopefully, the mobile agents meet other agents that are searching for the same topic. A publicist advertises a topic that it is looking for. A ferret looks for advertisers of a topic and provides the location of the information consumer. A guru schedules meetings with publicists and ferrets by remembering which agents they met and the topics they were interested in.

Recently, mobility has been proposed to monitor the health status of service providers subscribed with a registry or directory service in an autonomic computing environment [Thoma03]. A fault tolerant distributed directory service is responsible for maintaining information with respect to the health and functionality of each service provider. When a service provider is faulty, the registry is responsible for logging the abnormality. Consequently, clients only receive references to healthy service providers. Steady communication links between the registry and service providers are needed to monitor the health status. However, the distributed systems of today operate in an environment that has no defined boundary. This means that systems dynamically alter scale and connectivity during service provision. Furthermore, new functionality is added to services during service provision and resources are shared across organisations. Mobile agents have been proposed to visit the service providers periodically to retrieve information on the execution status. In the dynamically changing network, the mobile agent is responsible for finding the route to the service provider even if a link fails.

A further example is the use of mobile agents to gather load balancing information for web servers and redistribute jobs to servers with a lower load. A distributed pool of web servers can improve the quality of web services by replicating resources to deal with concurrent client requests and crash failures. A client request can be sent to alternate servers according to load balancing strategies. Load balancing aims to distribute client requests evenly to each server. According to [Cao03] traditional load balancing strategies mix the load balancing policies with the service implementation. Consequently, maintaining the load balancing strategy is complex. Furthermore, servers must be frequently polled to gather load balancing information. If a server rejects a job then another round of polling must be enforced. Conversely, in [Cao03], mobile

agents are employed to separate the load balancing strategy from the service implementation. If a server rejects a job, then the mobile agent can dynamically select the next best strategy, thus reducing latency. Currently, experimental results demonstrate that employing mobile agents in a LAN environment produced better performance than the load balancing module employed by the Apache web server for a LAN environment. Mobile agents maintained the lowest deviation of load distribution (in terms of job queue length). Throughput (total client requests per second) was greater using mobile agents for concurrent client requests greater than five hundred.

From the above applications of mobility it can be deduced that mobile agents may be useful to report on the status of remote hosts autonomously to the user application. The advantage of the mobile agent strategy is to sense the network conditions and links at remote hosts and act accordingly. Furthermore, under some conditions, mobility can be useful to compress information stored at remote hosts. It appears that the trend of applications that use mobility is to provide mobility at either the system application level or as middleware. To summarise, mobility can be useful in applications to:

- Compress large amounts of information at remote hosts, e.g. filter the best buy for a product offered by a group of known suppliers.
- Gather state information while adapting to dynamic network conditions. For example, intelligent mobile agents can react to failed communication links or slow servers.
- Discover a network topology that changes frequently. This is applicable to adhoc networks in mobile computing environments where there are frequent connections and disconnections.
- Perform actions on behalf of a mobile computer user at a stable host during periods of disconnection. Later, when the user is reconnected, the mobile agent can be retrieved.

4 Mobile agent problems

This section highlights a brief overview of the problems with mobile agents and aims to identify the key technical and non-technical hurdles faced for acceptance in industry. Technical hurdles are understood as problems influenced by mobility. For example, security is an obvious concern when private code and data is shipped over a network link to remote hosts. Non-technical hurdles are high level issues that may cause concern for industry.

Perhaps a key challenge is identifying a universal definition for mobile agents. Indeed, this is highly unlikely to be resolved when the term *agent* is loosely defined. Section 1.4 highlighted the confusion that exists between mobile agents and mobile code. The two terms are often used interchangeably within the research community. Further confusion is added concerning which community mobile agents belong to. Some perceive that a mobile agent is synonymous to a distributed object that can migrate state and code autonomously across hosts. Others belong to

the distributed artificial intelligence community whereby mobility is viewed as a characteristic of an agent. In this context, a mobile agent possesses knowledge of its environment, executes to satisfy goals and communicates using an Agent Communication Language (ACL). Furthermore, the mobile agent is capable of migrating autonomously to another location. It is unlikely that there will be a true definition for a mobile agent. It appears that the current trend for researchers is to form their own definition and develop a solution to suit. Consequently, the lack of a concrete definition influences uncertainty amongst industry and researchers. Unfortunately, this can be a factor towards reluctance to use the technology.

There is significant debate concerning the characteristics of mobile agents that make them suitable to specific application domains. For industrial acceptance a comprehensive study is required. Application domains suited to mobile agents were discussed by leading researchers in [Milojivcic99]. Electronic commerce, network management, information retrieval and mobile commerce were identified as potential applications of mobile agent technology. Mobile agents were envisaged as useful for autonomously representing users. A user could dispatch a mobile agent to perform a task at a remote node, e.g. querying a database. Mobile agents also have potential for data-intensive applications where data is remotely located and the user has specialised needs. For example, mobile agents may be launched by mobile devices to provide personal mobility, i.e. a mobile agent may represent a personal user profile and execute at a remote location, contacting the user when events of interest occur.

Despite these potential uses, there appears to be no consensus concerning a killer application where mobile agents are used as the main structuring unit [Milojivcic99, Lange99, Schoder00]. It could be argued that traditional technology may equally be used. For example, client server technology could be employed for querying a remote database. Industry may prefer familiar technology and established methodologies. Mobile agents may therefore be perceived as a solution in search of a problem domain [Nwana99].

Although it is easy to conceptualise application scenarios where mobile agents may be useful, awareness of the proposed benefits is important. Benefits frequently claimed for mobile agents are clearly stated in [Lange99]:

- **Reduce network load**: Traditional distributed systems require multiple interactions to perform a given task, consequently increasing network traffic. Mobile agents can be transferred to remote hosts where large volumes of data can be processed locally.
- **Overcome network latency**: For critical real-time systems, where entities need to respond to environmental changes in real-time, the network latency involved is significantly large. Mobile agents are seen as offering a solution since they can act locally.

- **Encapsulate protocols**: Mobile agents can migrate to remote hosts to establish channels based on proprietary protocols. Conversely, traditional approaches require that hosts implement protocols to code output data and interpret input data. Consequently, it is difficult to upgrade protocol code.
- **Execute asynchronously and autonomously**: Mobile devices operate in environments with fragile network connections. Mobile agents can embed tasks that are released into the mobile network to execute asynchronously and autonomously.
- **Adapt dynamically**: Mobile agents can distribute themselves to hosts in an itinerary to maintain the optimal configuration for performing a task.
- **Naturally heterogeneous**: It is claimed that mobile agents are generally computer and transport layer independent. Indeed, this is true to some extent for mobile agent systems implemented in Java. However, a mobile agent may only migrate to hosts that provide the same agent server.
- **Robust and fault tolerant**: It is claimed that the ability of mobile agents to dynamically react to adverse events, e.g. host failure, facilitates robust and fault tolerant distributed systems.

The benefits of improved network load and latency are debateable. Indeed, for mobile agents to provide a significant network load benefit, investigations are required to understand the degree in which the size of the mobile agent overrides the performance benefit. Recall that a mobile agent is a computational entity comprising code and state. Indeed, some experiments have been performed [Picco98b], although it is felt that there exists no consensus. Equally it may be argued that network bandwidth is constantly increasing.

Interoperability among mobile agent systems is limited. Although standards have been introduced there are few mobile agent systems that are fully compliant. Indeed, Grasshopper [Baeumer03] is the only MASIF [Milojivcic98] compliant mobile agent system. Furthermore, the ability of mobile agents to migrate autonomously to hosts can be equally viewed as a hindrance to fault tolerance. For example, tracing mobile agents can be a problem. In this case, mobile IP can be employed whereby the home agent server is notified of the mobile agent's new location after migration. However, it is possible that the mobile agent continuously migrates during the latent period of notifying the home agent server of its new location [Murphy99].

Another serious concern for mobile agents is security. Customers are wary of trusting third party service providers. For example, a malicious host may modify the state of the mobile agent. From another perspective, third party server providers may be wary of granting execution privileges to foreign mobile agents. It may be the case that malicious mobile agents present denial of service attacks dramatically consuming CPU load. Furthermore, Schoder *et al.*

[Schoder00] perceive the absence of a social and legal framework as a challenge for controllable deployment of mobile agents. Problems such as stability and communication security are scarcely solved by a centralised control system. This is due to the number of mobile agents that dynamically join the system in addition to the technical constraints of the underlying communications networks.

Essentially, mobile agents should be perceived as a uniform solution to many problems, such as network bandwidth, rather than a new technology that provides services that are not possible using other technologies. This argument dates back to [Chess95], i.e. there are few overwhelming advantages and an equivalent solution can be found that does not require mobile agents.

5 Summary

This chapter has presented an overview of the current research within the mobile agent community.

The mobile agent community is a wide and constantly evolving research field. Subsequently, there is typically a lack of widely accepted terms and methodologies. The agent community lacks a concrete definition for the term *agent*. Furthermore, there exists terminological and semantic confusion for distinguishing between mobile code and mobile agents. This chapter has therefore presented a definition for the terms, *agent* and *mobile agent*, which are subsequently adopted by the remainder of the thesis.

There are two levels of interest with respect to the architecture of a mobile agent system, i.e. the mobile agent and the agent server. Consequently, this chapter has outlined techniques for a mobile agent to organise its travel plan and communicate with resources at remote hosts. Furthermore, the core mechanisms of mobility, mobile agent communication and interoperability have been described.

Finally, the chapter presented a summary of the motivation for mobile agents and highlighted a brief overview of the problems with mobile agents. The following chapter provides an introduction to exception handling and investigates the challenges for exception handling in mobile agent systems.

Chapter 3 Exception Handling and Fault Tolerance

1 Introduction

Businesses and society are increasingly dependent on *software systems*, increasing demand for their flexible evolution and *reliability*. Safety critical systems and the financial sector are typical examples of *dependable* software applications. The dependability of a software system is defined in terms of trustworthiness such that reliance can justifiably be placed on the service it delivers [Laprie85, Laprie92]. Reliability is an attribute of dependability, dealing with continuity of service [Laprie85, Laprie92]. This section defines a system model and exception handling framework to form a groundwork for subsequent sections. A software system (Figure 3-1 [Xu00]) is described as a set of interacting components co-operating to meet the demands of a system environment [Anderson81]. A system's external behaviour is the combined activity of its components. Design components manage component interactions and connections with a *system environment* that provides inputs and receives outputs. An input from the environment triggers a service in a component. A system undergoes a sequence of *internal states* influenced by component interaction. An internal state consists of design data values, component output values (*external states*) and the values of variables maintained directly by design [Garcia01]. Specific sequences of component interactions influence erroneous states and transitions. A component fault leads to component failure and a design fault leads to design failure. Both influence an erroneous transition, termed the *manifestation of a fault*. A *fault* produces one or more *errors* which in turn introduce *system failure*.

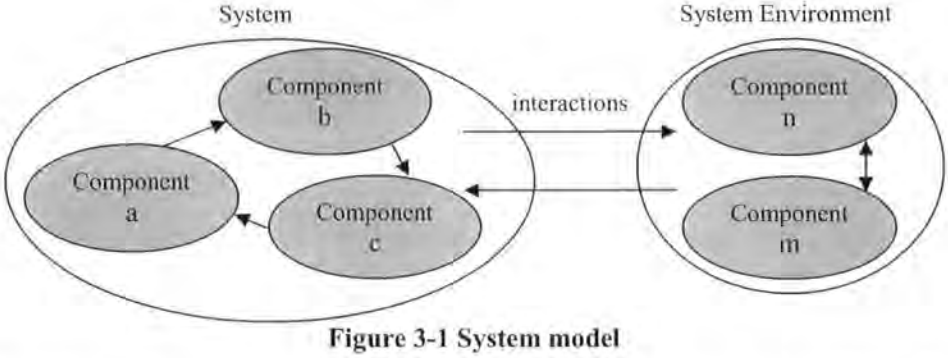


Figure 3-1 System model

Software engineers use a specification to assess system behaviour and reliability. A system failure is a deviation from the specification. An error is an invalid system state that could lead to failure. The source of an error is a fault that manifests itself in a software or hardware component. Faults and errors are irreversible. The presence of an error indicates a fault. However, a fault does not imply an error, unless observed. Duration and phase are two properties used to categorise faults presented in [Laprie92, Jalote94]. *Transient* faults are

temporary; failures and errors are apparent only within the fault duration and are consequently difficult to detect. *Permanent* faults are ever lasting. *Design* faults occur during system design or modification. *Operational* faults occur during system usage.

If faults can be prevented, it seems reasonable to assume that reliability is massively increased. Although this is true to some extent, it is unlikely to succeed, due to the intuitive and creative nature of design. Design faults may remain undetected, thus it is impossible to eradicate all faults. A mechanism is required to provide reliability in the presence of faults. This is termed *fault tolerance* and introduces redundancy to address faults. A system is fault tolerant if the behaviour of the system, despite the failure of some of its components, is consistent with its specifications [Jalote94]. Anderson and Lee [Anderson81] divide fault tolerance into four phases to manage the additional complexity and consequent increase in system state:

1. **Error detection**: Identify errors in system state.
2. **Damage confinement and assessment**: Assess the scope of error propagation and take appropriate measures for confinement.
3. **Error recovery**: Correct erroneous state to allow resumption of normal activity.
4. **Fault treatment and continued service**: Prevent a fault from immediately recurring to allow the system to provide the services outlined in its specification.

Error recovery is divided into *forward error recovery* and *backward error recovery*. Forward error recovery techniques correct an erroneous state to produce a new state that is, hopefully, error free. Backward error recovery techniques restore the current erroneous state to a prior error free state. Forward error recovery is application specific requiring prior knowledge of errors. Consequently, damage assessment and error prediction are significantly important. Backward error recovery replaces the entire system state, thus invalidating the need for damage assessment and error prediction. Backward error recovery provides higher reliability since unanticipated faults may be handled. However, there is the performance drawback of complete state restoration. Forward error recovery techniques use *measures*. Backward error recovery techniques use measures or *mechanisms*. Anderson and Lee [Anderson81] define a measure and mechanism as follows:

> "*A measure is a construction within the design of a system (e.g. in the program of an interpreted system) intended to perform a specific task. A mechanism is a construction within an interpreter which provides a specific facility*". [Anderson81]

Exceptions and *exception handling* provide a framework for the four phases of fault tolerance outlined above. Xu and Randell [Xu00] declare an exception as an abnormal response from a component or an abnormal event occurrence within a component. Figure 3-2 [Garcia01] illustrates the framework in the context of a system component that functions as both a server and client. Servers supply services requested by one or more clients and return a response. A *normal response* is the expected service output. An *abnormal response* signifies the presence of a fault using *interface, failure* and *local* exceptions [Garcia01]. Interface exceptions signify an illegal client request. A server signals a failure exception when it cannot provide a requested service. Local exceptions are raised when a component invokes its own fault tolerance measures. At this point there is an important distinction between *raising* and *signalling* exceptions. Raised exceptions are internal to a component. Signalled exceptions are communicated to clients. Consequently, fault tolerant measures are external to the component. A component's behaviour is clearly separated into *normal* and *abnormal (exceptional)*. Normal behaviour relates to the provision of services and is resumed upon successful exception handling. A component activates its abnormal behaviour upon receipt of an external (signalled) exception or during the occurrence of a local (raised) exception.

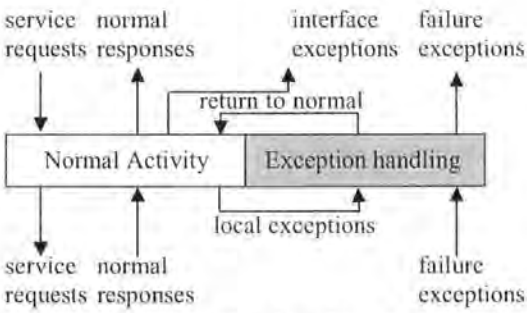


Figure 3-2 Exception handling framework

Abnormal behaviour therefore relates to a component's fault tolerant activity that is managed by *handlers*. A handler provides error recovery for a specific, anticipated, exception. An *exception mechanism* automatically detects *implicit* and *explicit* exceptions, searches for a suitable handler and transfers control to it. Implicit exceptions are detected at run time by the underlying hardware, language or operating system. Explicit exceptions are defined by the user and detected by the application [Garcia01]. Handlers are attached to a *protected region*, i.e. a bounded area of code that has the potential of producing exceptions.

Having established a system model and exception handling framework, it is now possible to define program control flow in the presence of exception handling. An exception raised within a protected region directs control to the component's abnormal activity. If the exception is local to the component the exception mechanism searches for a local handler. Alternatively, the exception mechanism raises an external exception (interface or failure) and signals it to the client component. Control reverts to the component's normal activity after the exception

mechanism has executed the appropriate handler. Two important issues arise at this point: (i) *exception propagation* and (ii) *control flow continuation*. An exception is propagated (signalled) when no suitable local handler exists. Propagation is *explicit* or *automatic* [Garcia01]. Explicit propagation signals the exception to the immediate caller where it is handled or signalled to a higher level component. (Note that the same exception is not necessarily signalled.) A general exception is propagated when no suitable handler exists within the caller or the program terminates. When automatic propagation is employed, the exception automatically propagates up the calling hierarchy until a matching handler is found, i.e. handling is not restricted to the immediate caller. The system should resume normal activity upon completion of exception handling. If the exception handler successfully averted failure, then control should resume at an appropriate location in the software to provide a continued service. Alternatively, the exception handler terminates and signals a failure exception. This scenario illustrates two models: (i) *resumption* and (ii) *termination*. The resumption model resumes normal activity at the statement subsequent to the one that raised the exception. The termination model terminates the activity of the component that raised the exception. If the component is *strictly terminated* the entire program is halted and control is directed to the operating system. *Return termination* terminates the signalling component and directs control to the statement that follows the protected region. Alternatively, the signaller may be terminated and retried. Resumption implies saving state before exception handling. Termination requires that the computation performed before an exception occurrence is undone and retried. Anderson and Lee [Anderson81] maintain that the termination model is preferred due to simple semantics, i.e. an exception is regarded as an abnormal event. Furthermore, the resumption model is highly likely to introduce failure since control reverts back to a potentially faulty component. This section concludes with a significant point raised in [Jalote94]:

> "*Note that exception handling provides a framework which supports the design of fault tolerant software, it is not a technique for fault tolerance. Fault tolerance has to be programmed using the primitives provided by exception handling*". [Jalote94]

In other words, exception handling is not a technique. It is a tool for the design and construction of fault tolerant software. In reality, excluding safety critical systems, exception handling is not considered a design issue.

So far, a system model, reliability and fault tolerance have been introduced. The system model was extended to describe an exception handling framework, forming a basis for outlining exception handling control flow. Sections 2 and 3 describe exception handling for serial and concurrent systems respectively.

2 Exception handling in serial systems

This section examines *software fault tolerance* techniques for serial systems. Software faults occur during software design and construction [Anderson81]. A serial system executes at a fixed location and has a single thread of control.

Abstraction is an established technique for controlling system complexity by recursively partitioning a software system into a hierarchy of modules. Each module consists of state variables and procedures. A module has an *internal state* and an *abstract state*. The internal state is the sum of all state variables. The abstract state is the result of applying a *procedure*. A procedure provides a specific coherent service, and one or more exceptional services (handlers), to handle abnormal situations. Figure 3-3 illustrates a module hierarchy.

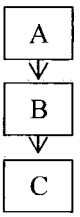


Figure 3-3 Module hierarchy

Modules are represented as nodes. An arrow from node A to node B signifies that A is a user of B, i.e. A invokes a procedure in B. Successful completion of A depends upon the outcome of the procedure invoked in module B. If a module is unable to perform a requested service an exception is raised and a handler performs error recovery. Ideally, the handler masks the exception from the calling module. In this case the procedure satisfies the intended service and returns a normal response. If the exception cannot be entirely masked from the caller, it is propagated (signalled). Figure 3-4 highlights exception propagation in serial systems. Explicit propagation occurs when no suitable handler exists in the signalling module. In this case the exception is propagated to the immediate caller. If a suitable handler exists remedial action is

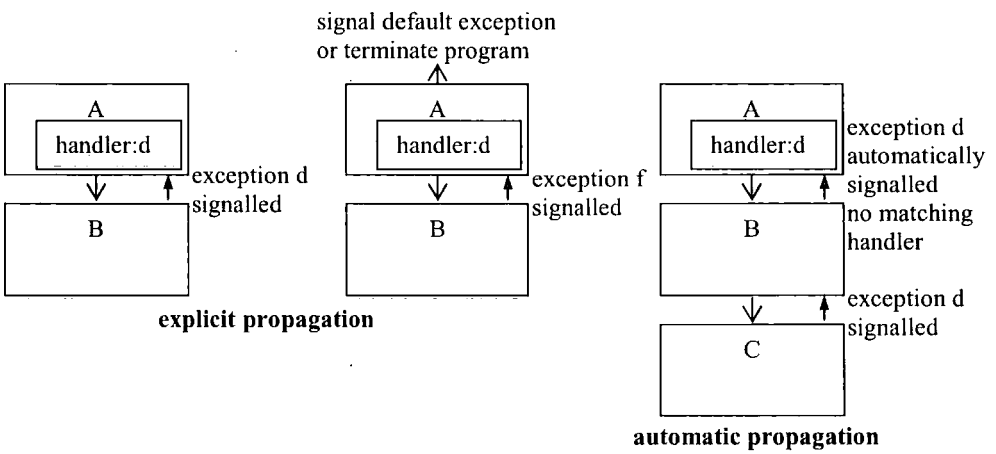


Figure 3-4 Exception propagation

taken. Alternatively, a default exception is automatically forwarded to the immediate caller or the program can terminate. Automatic propagation signals the exception up the module hierarchy until a suitable handler is located.

Normal activity should resume upon successful exception handling. Continuation of control flow may follow the resumption or termination model. The resumption model resumes control at the statement subsequent to the one that signalled the exception. Essentially, control resumes at the signalling module. The termination model forbids continuation within the signalling component. Garcia *et al.* [Garcia01] identify three divisions:

- **Return**: The signalling module is terminated and control resumes at the statement subsequent to the handler.
- **Strict**: The entire program is terminated and control is diverted to the operating system.
- **Retry**: The signalling module is terminated and retried.

By far the most difficult faults to eradicate are design faults. Even after a component has been designed, implemented and tested, there remains a strong probability that design faults are undetected. Consequently, this section aims to introduce briefly error recovery for software design faults, providing a basis for section 3.

So far, forward and backward error recovery techniques have been introduced. Forward error recovery fails to eradicate design faults since they must be foreseen at design. Unanticipated design faults are therefore unaccounted for. Alternatively, replication is an established technique for increasing the availability and fault tolerance of distributed systems. Essentially, multiple copies of a component are maintained. However, replication also fails since each replica depends upon the same design. Backward error recovery is preferred for unanticipated errors since the effects of execution are cancelled to a state that is, hopefully, error free. However, design faults still remain.

Two solutions exist: (i) *design diversity* and (ii) *data diversity*. Eradicating design faults implies using two or more components (variants), each implementing alternative and independent designs for the same specification. Subsequently, the probability of a common mode of failure is reduced. This solution is termed design diversity. Xu and Randell [Xu00] propose a framework (Figure 3-5 [Xu00]) whereby an adjudicator is a decision algorithm that

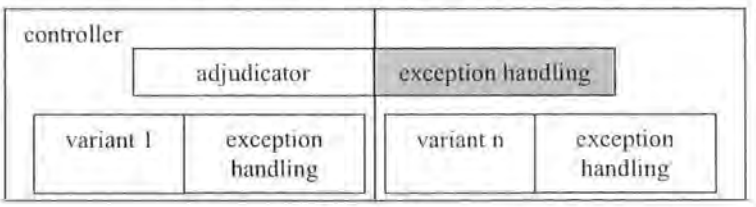


Figure 3-5 Design diversity model

serves to provide an error free result among executed variants. A controller manages the execution of variants. If a variant fails to produce an acceptable result, the controller selects and executes an alternative design.

Data diversity perceives design faults to be triggered by specific inputs. Consequently, if the inputs are varied, design faults may be eradicated. This approach executes an algorithm for logically equivalent sets of data. A decision algorithm is applied to determine system output according to the results. Data diversity is dependent upon data re-expression, i.e. reassigning data values that are logically equivalent. The focus of this thesis is therefore design diversity, namely recovery blocks [Horning74] and N-Version programming [Chen78].

2.1 Recovery blocks

A recovery block comprises an *acceptance test*, a *primary module* and a sequence of *alternate modules*. An acceptance test (adjudicator) is an error detection measure complementary to system state assertions and hardware error detection mechanisms [Xu00]. Tests are performed subsequent to server computation of results and prior to forwarding output to the client. A block comprises a series of alternate algorithms (variants), each considered an ideal fault tolerant component. Essentially, this means that each component has its own fault tolerance capabilities. The primary module executes first and is organised so that it is more desirable than its alternate modules. Subsequent alternatives employ increasingly degrading and simpler implementations. An alternate module is executed when the primary module/alternate fails the acceptance test or an exception is raised by another alternate module. Consequently, a recovery block scheme supplies *gracefully degrading software* [Anderson81].

Figure 3-6 [Xu00] illustrates recovery block control flow. Upon entering a recovery block the system state is check-pointed to enable backward recovery. The primary module executes and an acceptance test evaluates the results. If the primary module completes satisfactorily the recovery block exits. Alternatively, if an exception is raised, then backward error recovery is invoked to cancel the effects of the primary module and allow alternate execution. The next alternate module executes and an acceptance test is performed. The sequence continues until an alternate module passes the acceptance test or all alternate modules fail. In this case, a failure exception is signalled to the enclosing environment.

The recovery block scheme is largely extended. For example, the deadline mechanism [Campbell79] introduces support for real time systems to assume a predetermined maximum response time for services. A distributed recovery block scheme [Kim84] is capable of recovering from both hardware and software faults by applying distributed processing.

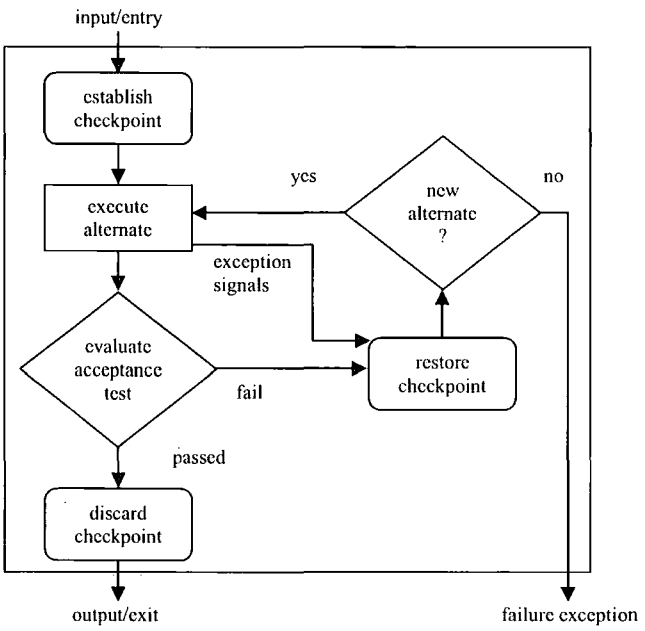


Figure 3-6 Recovery block control flow

The success of recovery blocks depends upon the adopted error detection mechanism and acceptance tests. Further design faults are likely to be introduced as the accuracy and consequent complexity of acceptance tests rises. Performance is also an issue, the worst case being when all alternate modules fail.

2.2 N-Version programming

N-Version programming (Figure 3-7) was founded by Chen and Avizienis [Chen78]. N-versions (N>1) of a program are implemented, each independently designed and executed. All versions are supplied with the same inputs and initial conditions.

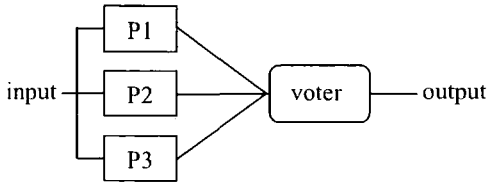


Figure 3-7 N-Version programming model

A *driver program*, synonymous to the controller in Xu and Randell's model of design diversity [Xu00], is responsible for invoking all versions, assimilating results and applying a voting algorithm to determine a single consistent result. The performance of N-Version programming systems is always equal to the worst case module, i.e. the longest executing. This has the advantage that execution time is predictable.

2.3 Design diversity costs

If, in an ideal world, resources were infinite, then multi-version software would appear to be a viable approach for producing dependable software. However, this is not the case in industry. It is debatable whether it is cost effective to channel development costs into a single version application with the hypothesis that increased development time provides dependable software. Should limited resources be applied to a multi-version application? Such a debate can only be answered provided that the cost of failure is known *a priori*. For example, in the consumer electronics industry recall costs are high. Conversely, software development costs in industries such as fly-by-wire aircraft, only account for a small proportion.

Newcastle University conducted experiments to determine the cost effectiveness of design diversity as opposed to single-version software applications [Xu00]. The experiments compared the costs for building a multi-version (3-version) and a single-version application for a factory production cell (presented in [Xu00]). Both systems were constructed using equal resources, each version being divided into equal units of time. The single version application was allocated three units of time. The results concluded that single-version software is more dependable since less faults and failures were detected. The multi-version application could equally be interpreted as being more reliable since there were fewer undetected failures. It cannot be said that single-version software is more dependable and safer in the presence of limited resources, since we cannot guarantee that the quality of software increases with time [Xu00]. Some failures will always remain undetected, largely due to contributions from the software crisis and design.

3 Exception handling in concurrent systems

The components of a concurrent system (Figure 3-8 [Xu00]) are *objects*, *threads* and *actions*. An object is defined by its state and behaviour (methods). Threads are active entities that invoke object methods. An action is a programming abstraction that allows the application programmer to group a set of operations on objects into a logical execution unit [Xu00].

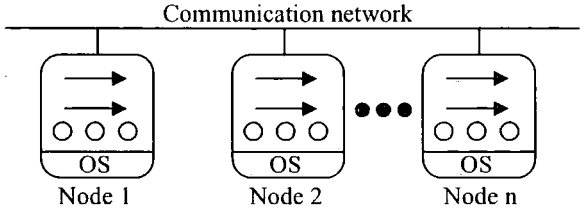


Figure 3-8 Concurrent system model

Anderson and Lee classify concurrency as *independent*, *competing* or *co-operating* [Anderson81]. The simplest form of concurrency is independent, whereby the activity of each process is isolated. Independent concurrency is present when an object is only accessible to one thread. Competing concurrency exists when an object is accessible to many threads purely for

resource consumption. No communication exists between threads. This thesis focuses on co-operative concurrency, i.e. threads have shared access to objects enabling inter-thread communication. Shared objects instigate the need for alternative exception handling mechanisms due to *thread interdependencies* and *concurrent exceptions.*

Thread interdependencies increase the likelihood of erroneous information propagation. Damage confinement is vital. Consequently, the handling of an exception should involve all co-operating concurrent threads [Garcia01]. Process dependencies, e.g. complex interactions, require several processes to handle a raised exception co-operatively. System state is consequently increased and likely to be inconsistent. Atomic actions are widely accepted for controlling error propagation and recovery in concurrent systems. Anderson and Lee [Anderson81] define an atomic action as:

> "*The activity of a group of components constitute an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity*". [Anderson81]

An atomic action is a structural unit that organises the interactions for a group of components. Interactions are temporal. Therefore an atomic action reflects the dynamic structure of a system. Error propagation is controlled by the restriction that interactions are internal. If an exception is raised during the activity of an atomic action then only those components participating need to be recovered. Externally, the execution of an atomic action is indivisible, i.e. the action appears as a primitive operation that transforms system state. Intermediate state changes are concealed. The use of atomic actions in serial systems has been introduced in the form of recovery blocks. Concurrent systems employ conversations, transactions or the hybrid CA action approach for atomic action mechanisms, in addition to an exception resolution mechanism.

3.1 Exception resolution

Serial systems raise a maximum of one exception. Concurrent systems alter this property when parallel processes simultaneously raise different exceptions. Which one is handled? Concurrent exception handling employs an exception resolution mechanism to resolve simultaneous exceptions and determine a generic handler. Xu *et al.* [Xu00b] maintain that concurrent exception resolution must be addressed for the following reasons:

- Immediate communication of an exception among participating processes is difficult. Distributed systems have a higher probability that further exceptions will occur before participants are notified of an exception occurrence.
- Concurrent exceptions may trigger a further serious fault.

- Distributed systems are inherently more complex to design compared to centralised systems. Consequently, an increase in the frequency of design faults is witnessed.
- If concurrent exceptions are ignored, all participants must co-operate for error detection and recovery. Consequently, performance is hindered.

Campbell and Randell proposed an exception resolution tree [Campbell86]. Exceptions associated with an atomic action are arranged into a tree hierarchy, whereby a higher level exception has a handler capable of dealing with lower level exceptions. If exceptions are raised concurrently, the resolving exception is the root of the highest subtree that contains all exceptions. Such an approach is static, whereby the designer must anticipate application exceptions. Another static approach is the chain algorithm [Jalote86] that employs both forward and backward recovery. Processes are statically linked and employ a rendezvous as the communication link. Each process receives exceptions from its left neighbour and forwards the resolution to the right. The rightmost neighbour resolves the exception and transmits results to the left that then calls the appropriate handler. An implementation for an exception resolution mechanism in Ada 83, Ada 95 and Java is presented in [Romanovsky00] that uses a controller process to collect all exception information from participants and co-ordinate resolution.

The exception tree is accepted as the most suitable approach [Campbell86, Romanovsky00] since it is inherently object-oriented. However, design complexity is increased, i.e. a higher level handler must be capable of providing recovery implemented by lower levels. Furthermore, the resolution tree can be built only when the designer is clear about the errors that are to be tolerated and the handlers can be implemented only after the tree has been built [Romanovsky00].

3.2 Conversations

The conversation scheme [Randell75] aims to provide co-ordinated error recovery for a group of interacting processes. Interactions may occur through message passing or referencing shared objects. A conversation (Figure 3-9 [Xu00]) is an application unit comprising a *recovery line*, a *test line* and two *firewalls*. The recovery line, established before interaction, is a series of co-ordinated checkpoints employed by interacting threads to enforce backward error recovery.

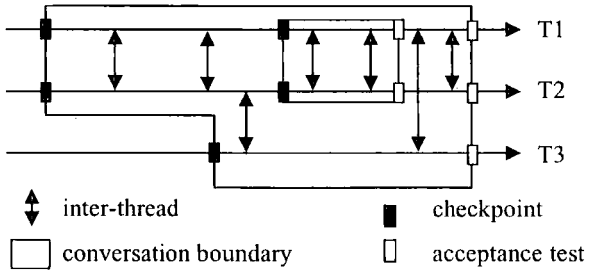


Figure 3-9 Conversation

A test line consists of acceptance tests for each participating thread. The firewall imposes the restriction that a thread may only communicate with conversation participants. A recovery mechanism is triggered if a participant raises an exception.

Backward and forward recovery techniques may be employed by the exception handlers. If backward recovery is used, all participating threads roll back to the check-pointed state and retry, possibly executing an alternate algorithm. Forward error recovery is useful for errors that affect the environment, e.g. hardware devices, users etc., since they cannot backtrack. The conversation is only successful when all processes satisfy acceptance tests at the test line. In particular whenever recovery is co-ordinated, the domino effect [Randell75] is avoided. The domino effect occurs in the presence of backward recovery and process communication. It is possible that rollback may have an uncontrolled cascading affect, e.g. the state of all recipients must be rolled back if commands that involve message passing are undone.

Campbell and Randell [Campbell86] introduce exception handling mechanisms into conversations for asynchronous systems. A system is structured as a series of actions or conversations, each containing a group of co-ordinating processes. Each action owns a set of predefined exceptions. Participants specify handlers for all or some of the exceptions. When an exception is raised the appropriate handlers are initiated within all participants. When a co-operating thread has raised an exception, error recovery should proceed in a co-ordinated way [Garcia01]. If an exception is raised for which a component does not own a handler, an atomic action failure exception is signalled to the containing action. Garcia *et al.* [Garcia01] identify three scenarios when participants may leave the action:

1. No exceptions have been raised.
2. An exception has been raised and the called handlers have recovered the action.
3. There are no appropriate handlers or recovery is not possible.

The resolution mechanism employs an exception tree to resolve different concurrent exceptions. The root of the smallest subtree containing all concurrently raised exceptions is selected as the resolving exception. Nested actions introduce the scenario whereby a thread raises an exception simultaneous to participant activity in nested (internal) actions (Figure 3-10).

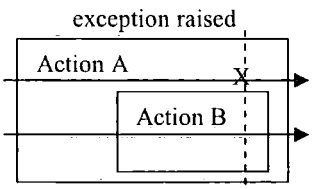


Figure 3-10 Nested action activity

In this case, the internal atomic actions are aborted before the containing action invokes its own fault tolerant activity [Campbell86]. Each action defines one and only one *abortion exception* that is raised when the containing action signals an exception. Participants in the containing action suspend, while the nested action applies measures to abort itself and consequently terminate. Only then may the containing action handle the exception.

3.3 Co-ordinated Atomic Actions

The Co-ordinated Atomic Action scheme (CA action) [Xu95] integrates conversations, exception handling and *transactions*. Conversations provide co-ordination and backward recovery but do not support use of shared external resources [Xu00b]. The consistency of shared objects is controlled by *transactions* to guarantee the ACID (atomicity, consistency, integrity, durability) properties. A transaction may be considered as an action that encapsulates and performs a sequence of operations on shared objects [Xu95]. The effects of performing the transaction are written using a commit operation. An abort operation undoes the effects of a commit by employing backward/forward error recovery. External objects are designed independently of participating CA action threads and are responsible for maintaining consistency in the presence of concurrent updates. Transactions mask the effects of concurrent updates and appear to be serially executed. The internal state of the transaction is invisible to others, i.e. execution appears as a primitive operation.

Exception handling now involves shared objects. If an exception is raised during the activity of a CA action then there must be a guarantee that shared objects are in a consistent state after recovery. Forward or backward recovery techniques may be employed. If any of the external shared objects fails to reach a correct state, a failure exception must be signalled to the containing action [Xu00].

The semantics of a CA action are identified as follows:

- If backward error recovery is supported, a recovery line is established.
- Establish a test line and a global test for the entire action.
- Upon error detection, all participants co-operatively apply forward and/or backward recovery.
- Explicit co-ordinated error recovery is employed for internal threads.
- External objects must possess atomicity and supply their own error co-ordination mechanisms.
- Communication is restricted to internal participants and external shared objects.
- Accessing external atomic actions involves starting a transaction.

If the CA action is successful, the recovery line is discarded, all related external atomic objects are committed and the CA action terminates. Alternatively, if the CA action fails all participants roll back to a checkpoint and execute alternates. Any transactions must be aborted during backward recovery.

3.4 Open Multi-Threaded Transactions

Kienzle *et al.* [Kienzle01] employ transactions (Figure 3-11) as a system structuring unit and exception context. Threads can create or join a transaction at any time and may fork and terminate within a transaction. Self-checking, shared transactional objects provide a communications medium for participating threads. Self-checking is understood as employing preconditions, postconditions and invariants. If any of these are violated an exception is propagated to the calling participant. In this sense, thread communication is implicit and participants are loosely coupled and independent.

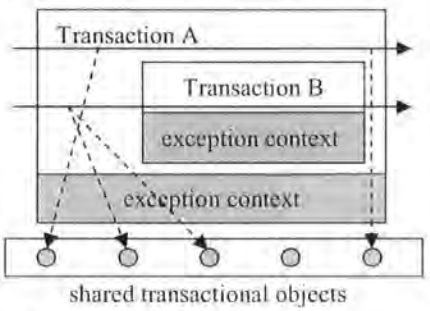


Figure 3-11 Open multi-threaded transactions

Exceptions are classified as internal and external. Each participant defines a set of handlers for the internal exceptions that may be raised for the duration of the transaction. When a participant raises an exception, the corresponding handler is invoked, thus completing activity. External exceptions are signalled explicitly outside of the transaction and trigger transaction abortion. If a participant aborts, i.e. an external exception is signalled, the entire transaction is aborted. Consequently, the exception is either signalled to the enclosing transaction or backward recovery is employed and the transaction is retried. Equally, forward error recovery can perform compensating state actions on transactional objects. All external threads must leave synchronously and the transaction commits only when all participants vote to commit.

It is claimed that open multi threaded transactions are suitable for dynamic systems whereby the number of transaction participants are unknown *a priori*. Each participant thread strives to handle an exception locally. Consequently, participants are autonomous in [Kienzle01]:

> "*Although they perform joint work inside a transaction and have a common goal, they are not tightly synchronised and can perform jobs even when not all of them are in the transaction*". [Kienzle01]

Open multi threaded transactions assume that each exception can be handled locally by the participant that raised it. Each participant is independent in the sense that communication is achieved solely through accessing transactional objects. However, communication is still necessary to signal exceptions among participants. Consequently, a resolution mechanism is controversially abolished. Participants enter the transaction with the same goal. If an exception is raised participants are likely to be affected. It appears that continuous exceptions will cause the transaction to abort. This results in a retry or an external exception that is signalled to the containing environment. This is a performance hazard since cascading exceptions occur until the transaction aborts. At this point, transactional objects are in an erroneous state. Clearly, the scheme is only scalable to applications that are not inherently co-operative.

4 Exception model for mobile agents

Exception handling is essential for mobile agent systems, as is the case for other software systems. A developer is particularly interested in the abnormal situations that may occur during execution. Armed with this knowledge, robustness is enhanced by employing redundant code to address errors and provide a continued service. Exception handling can be utilised at the design stage. Without this knowledge and exception handling the application will terminate whenever an error is encountered. Mobile agents are distinguished by the ability to autonomously migrate code and state between hosts. Consequently, further challenges are introduced that threaten the robustness of an application. Typical examples are mobility (node and communication link failures) and random interactions. Within an open environment security violations (inadequate access rights for host resources) also raise additional exceptions. So far, little attention is evident towards exception handling for mobile agents. This is despite the fact that previous experience has shown that exception handling is the most complex, misunderstood, poorly documented and least tested part of a software system [Parnas90]. An exception taxonomy is clearly useful as a first step to address exception handling in mobile agent systems.

A taxonomy may be multi-level or single-level. A multi-level taxonomy is detailed consisting of one or more sub classifications. The only exception handling taxonomy for mobile agents that is known, to date, is by Tripathi and Miller [Tripathi01] (see Table 3-1 next page).

A single-level taxonomy, selected by Tripathi and Miller [Tripathi01], comprises a single high level classification. Although a multi-level taxonomy may be complex and difficult to interpret there is the strong advantage that there exists a smaller degree of overlap between classifications. The taxonomy presented in Table 3-1 (see next page) concisely establishes a classification of exception scenarios within mobile agent environments. Such a benefit comes with the price of ambiguity. The mobility and security categories are linked. Assume that a mobile agent requests migration and subsequently relocates with insufficient access privileges.

Is the scenario classified as a mobility or security exception? A mobility exception is equally appropriate since the agent is unable to perform the task upon migration. A similar scenario could arise between the security and communication categories. An agent communicates with a proxy to access host resources. Resources are successfully allocated if the requesting agent has sufficient access rights. If the requesting mobile agent lacks sufficient access rights, is the scenario described as a security or communication exception? One approach would be to signal all exception classes that are relevant. However, further complication is introduced during design when the client component must ensure that all possible exceptions are caught. Certainly a measure of severity could be introduced into the taxonomy, e.g. those exceptions that result in termination. Furthermore, there exists some confusion between fault tolerance and exception handling. For example, message delivery failure is classified as an exception. However, this is treated as a fault tolerance issue by Murphy *et al.* [Murphy99]. Exception handling is a limited form of software fault tolerance, i.e. the operation that caused the fault can be ignored or a predefined degraded response is incorporated into the recovery handler. Conversely, a system is considered fault tolerant if the behaviour of the system despite the failure of some of its components is consistent with its specifications [Jalote94]. The effects of such failures are masked from the client. Therefore exception handling is not truly fault tolerant, since a departure from the specification is possible.

Exception	Description
Mobility	Occurs when a migration request is made to a non existent host, communication link failure or insufficient security permissions.
Security	Some examples include insufficient resources allocated for a task and illegal modifications of mobile agent state.
Communication	Examples include failure to locate receiving mobile agent and message delivery failure.
Co-ordination	Mobile agents fail to communicate within a bounded region of time.
Configuration	Occurs due to incorrect configuration of agent server, e.g. inadequate security privileges or incorrect location of classes utilised by the mobile agent.

Table 3-1 Exception taxonomy

Tripathi and Miller [Tripathi01] adopt their taxonomy for an exception handling architecture specific to mobile agents. This is based upon work done by Mark Klein [Klein99] for multi-agent systems. To the author's knowledge, these are currently the only approaches for exception handling specific to the agent paradigm.

4.1 Exception handling for mobile agents

Traditionally, co-ordinated exception handling has been "hard coded" into co-operative agents resulting in agents that are difficult to maintain and understand. Klein [Klein99] proposes a knowledge based exception handling architecture that clearly separates exception handling from an agent's normal behaviour. An exception handling agent monitors an agent for symptoms and suggests both a diagnosis and resolution using a heuristic classification process. An action and query language provide the interface between the agent and exception handling agents. Actions performed for recovery may involve reordering, adding or removing tasks. Recovery is a plan that completes slots by querying the state of the agent. Tripathi *et al.* [Tripathi01] adopt a similar framework for mobile agents. Exceptions are divided into two categories: (i) *internal* and (ii) *external*. An internal exception is handled independently by the mobile agent. External and unanticipated exceptions are resolved through co-operation with a group of agents via a guardian. The guardian is a central static agent that monitors application agents and provides global recovery. When a mobile agent encounters an external or unanticipated exception it notifies the guardian agent. Communication between mobile agents and their guardian may be achieved remotely or locally, i.e. the mobile agent can relocate to the guardian's environment. Exceptions are handled by communicating commands to mobile agents that, in turn, modify and query state. In some cases the environmental state may be modified. Example recovery commands include: retry, terminate and relocation. Similar to Klein's [Klein99] architecture, a guardian agent may encapsulate an exception handling pattern.

Klein's exception handling architecture [Klein99] is limited to static multi-agent systems. Mobile agents are dynamic, i.e. an agent can relocate to a remote host. Adopting the architecture for mobile agent systems would consequently reduce performance, i.e. each server must host a knowledge base that mobile agents use to register a model of their normal behaviour. Agents are introduced for detecting exceptions and determining a resolution. These are only necessary for the duration of an agent's stay and are justified providing an agent performs a sufficient task. The process is repeated when a mobile agent migrates to the next agent server in the itinerary. Distributing the knowledge base at hosts visited by agents is clearly costly in terms of maintenance. However, the architecture incorporates domain independent recovery patterns, i.e. exception handling strategies for specific abnormal events. Benefits include enhanced understanding and testability, although agents must implement interfaces to report a behaviour model and enable modification of their actions. Domain independent recovery relies upon correctly identifying the causes of failure for each generic problem solving process in the knowledge base. An agent registers a model of its normal behaviour that is then matched with the set of generic processes. Applicable failure modes are thus derived. The recovery technique modifies an agent's actions appropriately based upon the knowledge of failure modes and their respective resolution strategy. The drawbacks are: (i) implementation is inherently complex to

account for generic exception handling, (ii) there is no guarantee for correctly detecting and handling exceptions within any application domain and (iii) recovery is costly when designing resolution strategies and envisaging a suitable taxonomy of generic problem solving processes.

Knowledge of the application domain is needed to some extent for reliable recovery. Indirectly this is acknowledged in [Klein99]:

> "*An important characteristic of heuristic classification is that the diagnoses represent hypothesis rather than guaranteed deductions: multiple diagnoses may be suggested by the same symptoms, and often the only way to verify a diagnosis is to see if the associated prescriptions are effective*". [Klein99]

An alternative is an exception handling framework that divides component activity into normal and exception handling behaviour. Normal activity corresponds to service provision. Exception handling is activated upon receipt of an external (signalled) exception or during the occurrence of a local (raised) exception. Forward recovery is employed whereby the system state is corrected in response to exceptions predicted for the application domain. Although application dependent, this approach is preferred since it is cheap and easy to implement. For example, language primitives exist, e.g. try-catch blocks and throw statements, to identify handler code and signal exceptions to a client component. Application specific exceptions are easily introduced through an inheritance hierarchy.

4.2 Exception handling for failure models

A failure model defines the ways in which failures may occur in order to provide an understanding of the effects of failure [Goulouris00]. Hadzilacos and Toueg [Hadzilacos94] provide a failure model (see Table 3-2 next page) that distinguishes between process and communication channel failures for distributed systems. In order to establish an exception handling framework for mobile agents, a failure model is required to provide an understanding of the likely ways in which a mobile agent system fails. Only then can measures be taken to address failures. So far, there exists no failure model for mobile agent systems [Waldo01]. Pleisch and Schiper [Pleisch00] provide an insight into failures that are pertinent to mobile agent systems. Mobile agents can operate within a synchronous and asynchronous environment. Mobile agent systems that operate within an asynchronous environment have no boundaries on message passing and communication delays. The root cause of mobile agent failure is therefore a problem in asynchronous environments, since it is difficult to distinguish between a slow communication link and a mobile agent that fails by crashing. If a crash is diagnosed, it may be the case that the mobile agent is delivered on a slow machine or communications link, resulting in duplicate mobile agent execution. Conversely, blocking occurs if it is assumed that the

mobile agent will eventually be delivered due to a slow communication link, when in actual fact it crashed. Indeed, several scenarios exist for a crash failure. A mobile agent executes at an agent server. An agent server is a process that runs at each host visited by the mobile agent.

The following relationships exist between these three abstractions for a crash failure:

- **Host crash**: Affects mobile agents and agent servers.
- **Agent server**: Mobile agents are lost.
- **Agent**: Machines and agent servers are not affected.

Failure class	Failure	Effects	Description
omission			A processor or communication channel fails to perform actions expected.
	fail-stop	process	Process remains halted. The failure is detected by other processes and stable storage is unaffected.
	crash	process	Process remains halted. Other processes may not detect such a failure.
	omission	channel	An outgoing message, within a send buffer, is not received at the destinations incoming buffer.
	send omission	process	Process completes performing a send but the message is not assigned to the send buffer.
	receive omission	process	A message arrives in the process' receive buffer but the process does not receive it.
arbitrary			Failure of a process or channel is arbitrary, i.e. messages may be transmitted randomly, channel omissions may be committed or a process may deliberately halt.
	timing	process or channel	Occurs in synchronous distributed systems where bounded time limits are exceeded for execution time, message delivery and clock drift rate.
	clock	process	The local clock exceeds drift from real time.
	performance	process	Process exceeds time interval between performing two computations.
	performance	channel	Message transmission exceeds boundary.

Table 3-2 Failure model for distributed systems

A failure model is needed to gain understanding of the likely ways in which an application can fail. Only then can exception detection mechanisms, and consequently recovery, prove to be effective. Furthermore, a measure of severity is implicitly gained to determine a resolution strategy. For example, security exceptions may be handled using termination. A mobility exception, e.g. a non existent agent server in the itinerary, may be handled by dispatching the agent to the next host in the itinerary. Clearly a failure model will differ across application domains.

5 Summary

This chapter has investigated techniques for exception handling in traditional serial and concurrent systems. Subsequently, the chapter outlined the challenges for exception handling in mobile agent systems. These are namely distinguished by the ability of mobile agents to autonomously migrate code and state between hosts, thus leading to increased complexity with respect to error detection and confinement.

Finally, the chapter highlighted the fact that little attention is evident towards exception handling for mobile agents. An overview of existing approaches was briefly presented and highlighted the lack of a concrete failure model for mobile agent systems. A failure model is required to provide an understanding of the likely ways in which a mobile agent system fails. Only then can exception detection mechanisms, and consequently recovery, prove to be effective.

The following chapter presents a failure model for mobile agent systems and focuses on providing exception handling for server crash failures.

Chapter 4 The Mobile Shadow Exception Handler

1 Introduction

Chapter 2 discussed the mobile agent paradigm in general, identifying: what a mobile agent is, the architecture of a mobile agent system, applications of mobility and the current problems with mobile agents. A mobile agent is a computational entity that is capable of relocating its code and data to remote hosts to execute a task on behalf of a user. The sequence of hosts that a mobile agent visits is described by its *itinerary*. Weak mobility [Fuggetta98] is assumed, i.e. the mobile agent restarts its execution at each host. A remote host runs an agent server platform that provides an execution environment for the mobile agent. The *home agent server* is where the mobile agent is created. Remote hosts visited by a mobile agent are assumed to execute the same agent server platform. Chapter 3 then discussed fault tolerance and exception handling in general. In particular, exception handling was explored for mobile agents and concern was raised for the lack of an established failure model for mobile agent systems. The remainder of the thesis is aimed at exploring exception handling for mobile agent loss due to crash failures of agent servers. Furthermore, the thesis is concerned with how this can be used amongst a group of collaborating mobile agents.

Chapter 3 highlighted the lack of a failure model for mobile agents. To the best of the author's knowledge there are few failure models for mobile agents. A failure model is important to understand the ways in which a distributed system can fail. The developers of any fault tolerant system must design a failure model that outlines:

- Failures masked from the user.
- Failure semantics, e.g. does a process halt when it fails.
- Assumptions made regarding the environmental conditions of failure, e.g. there may be a boundary for the total number of failures tolerated.
- Protective mechanisms to mask failures.
- Level where fault tolerance is pitched, e.g. middleware or transparent to the application developer.
- System model of the software and hardware environment.

A system model of the software and hardware environment facilitates identifying the components that are susceptible to failure and the boundary, or coverage, for the effects of failures. In this thesis a model describes the entities or components that form the system and their interrelationships. The software environment is described at a high level of abstraction in

terms of the functional components and interrelationships. The hardware environment is also characterised by the model to a limited extent. For example, nodes in the network may be fully connected.

A mobile agent system can fail in many different ways. For example, a communication link may fail or a network partition could prevent migration. Alternatively, a mobile agent may not have sufficient access rights to execute at a remote agent server. This thesis explores a crash failure model for mobile agents. Tolerating crash failures is particularly important for mobile agents since a mobile agent represents a single point of failure. Without fault tolerance, an agent server process or host that fails by crashing has the effect that all local mobile agents are lost and consequently fail to return results to the home agent server.

The execution path of a mobile agent can be partitioned into stages (Figure 4-1). Each stage corresponds to an agent server in the mobile agent itinerary.

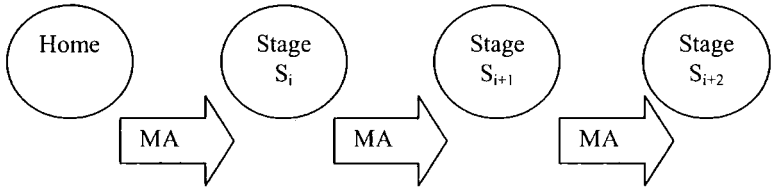


Figure 4-1 Mobile agent stage execution

There are three approaches to provide fault tolerance for mobile agents to survive agent server crash failures:

1. **Spatial replication**: Each itinerary stage is described by a set of agent servers that can potentially execute the mobile agent. A replica of the mobile agent is dispatched to each agent server. Agent servers in a stage monitor the executing mobile agent's agent server for failure. At the end of each stage, agent servers agree upon the agent server that executed the mobile agent and the set of agent servers for the next stage. Consensus is used for agreement. Once agreement has been reached the mobile agent is sent to the set of agent servers in the next stage.
2. **Temporal replication**: One or more visited agent servers monitor the current agent server. If an agent server crash occurs, a mobile agent is dispatched to the next agent server in the itinerary or an alternative agent server.
3. **Implementation language**: Crash failure handling is implemented in a scripting language. The scripting language is used to specify a tree of travel plan choices for the mobile agent. If the mobile agent fails at a host, it backtracks to the nearest parent and selects an alternative choice or defers the visit to the host. Hosts visited by the mobile agent send heartbeat messages to the immediate previous host.

Transaction protocols are sometimes employed between sender and receiver agent servers to provide *atomic migration*, i.e. the mobile agent is accepted and executes at its destination agent server or not at all in the event of an agent server crash. There are two cases for the transaction boundary:

- **Receive, execute and send**: A mobile agent has executed at an agent server only when it has been received, executed and then sent to the next agent server in the itinerary.
- **Send, receive, execute and acknowledge**: A mobile agent has executed only when the previously visited agent server has received acknowledgement of completion.

Not all mobile agent systems provide fault tolerance mechanisms against the crash of an agent server. Table 4-1 summarises the mobile agent systems that explicitly provide fault tolerance for the loss of a mobile agent due to an agent server crash failure. Each mobile agent system in Table 4-1 is named with the fault tolerance mechanism applied for surviving crash failures, e.g. spatial replication, temporal replication or implementation language. Furthermore, the following properties are outlined for each mobile agent system:

- **Fault tolerance location**: Is the fault tolerance algorithm located at the agent server or in the mobile agent?
- **Recovery assumptions**: Assumptions made regarding recovery of crash and communication failures. For example, do agent server crash failures and network partitions eventually recover?
- **Communication assumptions**: Assumptions made concerning communication. For example, is reliable messaging assumed for communication between mobile agents?
- **Transaction mechanism**: Is atomic migration provided, or is the loss of a mobile agent handled after a crash has occurred? Alternatively, a transaction mechanism may not be used to handle the loss of a mobile agent due to an agent server crash failure.

	Location	Recovery assumptions	Communication assumptions	Transaction mechanism	Description
Mole [Strasser99] spatial replication	agent server	host crash and network partitions recover	reliable messaging for voting	atomic migration	transaction and leader election to vote upon mobile agent executed

Net Pebbles [Mohindra00] temporal replication and language	agent server	none	none	rollback script state at visited hosts	alternative migration points specified using scripting language
FATOMAS [Pleisch00, Pleisch01, Pleisch03] spatial replication	mobile agent	recover from crash and link failures	reliable broadcast for consensus	abort / undo duplicate agent servers from false suspects	consensus for agent server executed, agent servers in next stage and mobile agent sent
JAMES [Silva00] temporal replication	agent server	host crashes and network partitions recover	reliable messaging	atomic migration	transaction with two phase commit between sender and receiver
Design for GMD FOKUS [DeAssisSilva00] spatial replication	agent server	network partitions recover	reliable messaging for events	atomic migration	distributed transactions and leader election
NAP: Tacoma [Johansen99] replication	agent server	fail stop[1]	rely on transport protocol	no transaction mechanism employed	linear reliable broadcast to send agent

Table 4-1 Fault tolerant mobile agent systems

For a group of co-operating mobile agents the use of replicas at alternative agent servers can increase the complexity of the agent design. Furthermore, there are also interoperability issues to consider. The location of the fault tolerance algorithm may be situated at the agent server or

[1] Fail stop [Schlichting83] semantics define that a process halts upon failure and other processes detect the failure. Furthermore, stable storage is unaffected by failure and is readable by other processes.

at the mobile agent. Situating the fault tolerance algorithm at the mobile agent increases its size. Alternatively, the fault tolerance algorithm could exist at the agent server. However, all mobile agent platforms must be willing to incorporate the new fault tolerance design. Furthermore, using replication for fault tolerance in the mobile agent paradigm is complicated due to the fact that, if the primary mobile agent replica fails, it is not independent of its environment. This is because mobile agents indirectly modify the environment by service invocations. If the mobile agent fails then the modified state of the environment remains. Conversely, a replica that crashes in a traditional distributed system results in the entire state being lost. Additional complexity is evident for fault tolerance in mobile agent systems since there is no natural instance that can monitor the mobile agent as it migrates autonomously to remote hosts. Conversely, in traditional distributed systems the client monitors the server at its static location.

An exception handling scheme to protect mobile agents against agent server crash failures, requires an exception handling model. This outlines a system model to describe how mobile agents interact with software services at remote agent servers. Furthermore, the control flow for raising and signalling exceptions between mobile agents and agent servers is outlined. The remainder of the chapter is structured as follows. Section 2 introduces an exception handling model for mobile agents. Section 3 then presents an overview of failures for mobile agent systems and outlines the failure model adopted for mobile agents to survive agent server crash failures. Section 4 describes a conceptual design that uses exception handling to protect mobile agents from agent server crash failures. Finally, section 5 describes an implementation of the conceptual design and section 6 provides a demonstration.

2 An exception handling model for mobile agents

A mobile agent can be regarded naturally as a software component. Figure 4-2 illustrates an adaptation of the exception model for software components presented in [Garcia01, Xu00] for mobile agents. The main architectural components are the agent server, mobile agent and software services. The agent server executes as a thread within the interpreter process. The mobile agent is assumed to run as a thread within the agent server. An agent server AG_j offers a set of services $S = \{s_1, s_2, \ldots, s_n\}$. A service s_i is a software component that a mobile agent manipulates by issuing method calls (service requests). A software component (i.e. an agent or a

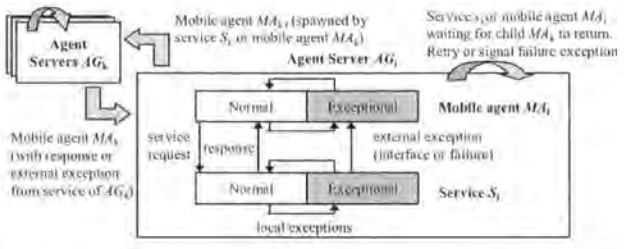


Figure 4-2 Mobile agent exception handling model

service) defines its own set of internal or local exceptions $I = \{e_1, e_2, \ldots, e_k\}$ and associated handlers $IH = \{h_1, h_2, \ldots, h_k\}$ that serve to provide corrective action. An internal exception occurrence e_i triggers the exceptional activity h_i within the software component. If the exception is successfully handled, normal activity resumes and completes, e.g. a service s_i completes its execution by providing a response to the mobile agent that made the service request. A mobile agent completes its activity by migrating to the next agent server in its itinerary. A mobile agent itinerary, *itin*, describes an ordered sequence of hosts that the mobile agent visits during its trip.

A corrective action is performed by a service or mobile agent in response to an internal exception. This is application specific and may involve dispatching a compensating mobile agent *CM* to interact with service s_j at a remote agent server AG_j. For example, a mobile agent may spawn a child to cancel a purchase made at AG_{i-1} and locate a cheaper product because it exceeded its budget.

A service s_i signals a set of external exceptions $E = \{interface, failure\}$ to a mobile agent when it fails to satisfy the service request. There are two classifications for external exceptions:

1. **Interface**: Input values supplied by the mobile agent violate the service specification.
2. **Failure**: The service is unable to provide a suitable response, e.g. a commerce service is unable to meet the delivery deadline for a given order.

Upon receiving an external exception the mobile agent may retry service s_i, locate a service at an alternative agent server or report back to the home agent server or parent mobile agent.

Figure 4-2 illustrates that the exception handling model is recursive. For example, service s_i at agent server AG_i may spawn a mobile agent MA_k to visit an agent server AG_k in reaction to a request made by MA_i. Similarly, mobile agent MA_i may spawn a child MA_k to perform a delegated task such as information retrieval. Consequently, the *owner* of a child is either a service or mobile agent. A mobile agent is dispatched a second time if it crashed or reported back to its owner with a failure exception. If the owner is a service a failure exception is signalled to the mobile agent that made the request, provided that the retry failed and no alternative service could be located. If the owner is a mobile agent, the failure exception is forwarded to its parent. The relationship between a parent and child is normally asynchronous. However, if the parent depends upon the results collected from its child, a synchronous relationship is introduced, i.e. the parent must remain stationary until its child has returned. For example, assume a mobile agent is dispatched to determine a purchase plan for PC system components, e.g. motherboard, CPU etc. The mobile agent dispatches a child to determine the best deal for a CPU. Due to hardware dependencies, the parent can only consider a motherboard when its child returns.

At this point it is necessary to discuss the failure model of the mobile agent for crashes. When a mobile agent crashes it may be the result of a host failure or a machine failure. Without fault tolerance the mobile agent is lost at the agent server that fails. The crash failure model depends upon the application. For example, if operations performed by the mobile agent are idempotent then there is no need for transactions since the mobile agent does not alter the state of the agent server. However, if the mobile agent modifies the state of the agent server indirectly through service invocation, then action must be taken to ensure that the mobile agent performs the necessary steps *exactly once*. Exactly once semantics are difficult to achieve in asynchronous distributed systems since it is has been proved impossible to distinguish between a slow agent server and a crashed agent server process. Furthermore, transactions are necessary to provide all or nothing mobile agent execution. Firstly, the session state for mobile agent execution must be durable or persistent. If an agent server crashes and later recovers, then the session state for each mobile agent must be restored from stable storage. If there is an error and the mobile agent cannot proceed, then it must be possible to roll back the session state of the mobile agent at that agent server. Section 3 defines the failure model adopted in this thesis.

3 Failure model

A failure model defines the ways in which failures may occur in order to provide an understanding of the effects of failure [Goulouris00]. Only then can recovery prove effective. A failure model is ideally defined by conducting field-based observations over a large time period for different systems in operation [Marsden02]. Information may be collected such as failure classification, frequency and the activities that lead to failure. However, it is believed that few studies exist for mobile agents. So far, there are few concrete failure models for mobile agent systems [Waldo01]. To the best of our knowledge the only existing failure model is presented in [Tripathi01]. Table 4-2 outlines a failure model for mobile agent systems.

The focus of this thesis is an exception handling scheme to protect mobile agents from agent server and host crash failures. Consequently, the scheme provides the foundation for the exception handling model to operate in the presence of agent server and host crash failures. Section 2 outlined a model for mobile agent exception handling to aid understanding the effects of crash failures. At this point, the failure model adopted for agent server and host crash failure is outlined to describe environmental assumptions and the effects of failure.

Mobile agents are assumed to operate in a synchronous network environment, e.g. a LAN. This means there is a maximum time boundary for migration and round trip time. Furthermore, network partitions, host crashes and communication link failures recover eventually. This last assumption is made by most of the fault tolerance schemes in Table 4-1.

Class	Description
Security	There are two types of attacker: malicious agent or host.
Malicious agent	Unauthorised access to services, denial of service (e.g. recursive cloning), and monitoring agent server and mobile agent activity.
Malicious agent server	Denial of execution, masquerading as a trusted agent server, state corruption, improper code execution and modifying system call results.
Communication	There are two classes of communication fault: transport and message.
Transport	Mobile agent transport failure, e.g. communication link failure or permission refused to execute at an agent server.
Message	Message failure due to dynamic location, e.g. out of sequence, duplicate or corrupted messages.
Software	Software faults within the mobile agent or a service at the agent server.
	Invalid inputs or response from a software service at the agent server.
	Denied access to a software service due to heavy load.
	Locking failures, e.g. deadlock and livelock.
Crash failures	Mobile agent fails to report back to the home agent server.
Mobile agent loss	Node crash due to hardware or systems software fault.
	Agent server crash due to operating system process deletion, system software failure or security attack. All active mobile agents and services at the agent server are lost.
	Mobile agent crash due to deadlock, node or agent server crash.
Application	Mobile agent blocks to communicate with child, or parent, that has crashed.

Table 4-2 Mobile agent system failure model

A mobile agent is assumed to fail under the following circumstances:

- **Agent server crash**: All local mobile agents and software services are lost.
- **Host crash**: The agent server process is lost in addition to all local mobile agents.
- **Communication link failure**: The mobile agent system transport protocol fails to deliver the mobile agent to the destination agent server.

Consequently, a crash failure denotes the occurrence of an agent server or host crash. This does not imply a mobile agent crash that occurs due to a software fault, e.g. an infinite loop.

An agent server AG_k hosts a set of mobile agents, $agents = \{ MA_1, MA_2, \ldots, MA_n \}$ and services $services = \{ S_1, S_2, \ldots, S_p \}$. Services are software components that mobile agents use to access resources at the agent server environment. A mobile agent MA_k migrates to the next

agent server in its itinerary, AG_{k+1}, as a result of a *go(AG_{dest})* operation, where AG_{dest} is the address of the next agent server in the itinerary, i.e. AG_{k+1}. There are three scenarios for failure when AG_k dispatches MA_k to AG_{k+1} at time t:

1. AG_{k+1} crashes at time $u < t$.
2. MA_k is accepted at AG_{k+1} at time u. AG_{k+1} fails by crashing at time $v > u$.
3. Communication link failure between AG_k and AG_{k+1} at time $u > t$.

In the first case the sender throws a failure exception when agent server AG_{k+1} crashes before dispatching MA_k. In the second and third cases, without fault tolerance, the mobile agent is lost.

An agent server crash is assumed to obey *fail stop* [Schlichting83] semantics and occurs due to failure of the Java Virtual Machine or a host crash. This means that all local mobile agents and services at AG_{k+1} halt. Furthermore the dispatching agent server AG_k eventually detects the loss of a mobile agent, due to communications failure or destination agent server crash. Eventually, it is assumed that agent server AG_{k+1} recovers and restarts the software services. However, the agent server, AG_{k+1}, is not responsible for restarting agents that were lost due to a crash failure. Consequently, the stable storage property of fail stop semantics [Schlichting83] is not required.

Furthermore, it is assumed that the mobile agent executes at agent servers for information retrieval only. In chapter 2 it was seen that distributed information retrieval is a popular application of mobile agents for collecting data at remote hosts and investigating the network topology. Consequently, the failure model adopted by the thesis assumes *at least once* application semantics. This means that a mobile agent may perform a task at an agent server at least one time, but possibly more. This is a reasonable assumption for information retrieval applications since a mobile agent does not modify the state of the agent server, i.e. interactions between the mobile agent and agent servers are idempotent.

Finally, it has been established that the failure model is for crash failures in a synchronous network environment. If the failure model was implemented in an asynchronous network environment then false failure suspicions are likely to occur since there is no time boundary for migration between agent servers and the total round trip time to complete the itinerary. Consequently, it is impossible to distinguish between a mobile agent that is lost due to an agent server crash and a slow communication link or processor. However, if the operations performed by mobile agents at remote agent servers are idempotent then the damage is confined to duplicate agents executing at agent servers.

4 The mobile shadow scheme

The mobile shadow exception handling scheme employs a pair of replica mobile agents, master and shadow, to survive agent server crashes. It is assumed that the home agent server is always available to its mobile agents. For example, if the home agent server crashes its mobile agents may return to a replica agent server. The *master* is created by its home agent server, *home*, and is responsible for executing a task T at the hosts described in its itinerary. Initially, the master spawns a shadow, $shadow_{home}$, at its home agent server before it starts the itinerary at AG_i.(Figure 4-3 a). Before the *master* migrates to the next host in the itinerary, i.e. AG_{i+1}, it spawns a clone or $shadow_i$ and sends a *die* message to terminate $shadow_{home}$(Figure 4-3 b). The $shadow_i$ repeatedly pings agent server AG_{i+1} until it receives a *die* message from its master. The functionality of shadow and master roles is now discussed for exception handling.

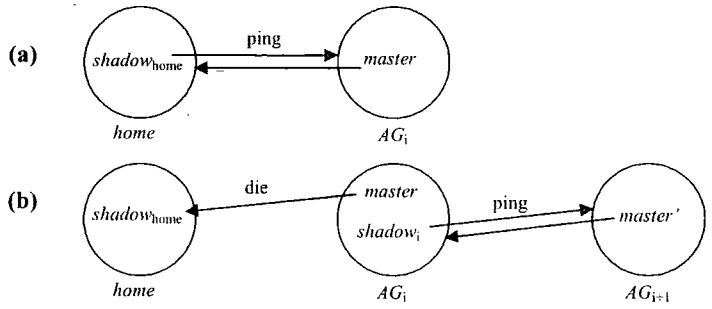


Figure 4-3 Normal execution for the mobile shadow scheme

A *shadow* is a clone of the master that acts as an exception handler for a master crash. The shadow pings its master's agent server. If a shadow detects a master crash it raises a local exception to signify master failure. The exception handler skips the master's current location and migrates the shadow to the next agent server. A shadow terminates when it receives a *die* message from its master. This signifies that the master has completed execution at AG_{i+1} and spawned a new clone $shadow_{i+1}$ to monitor agent server AG_{i+2} (Figure 4-4 a). However, assume the master is lost due to an agent server crash at AG_{i+1}. For example, AG_{i+1} could crash before the master migrates or during execution. The shadow, $shadow_i$, at AG_i detects the crash of its master, spawns a new clone $shadow'_i$ and proceeds to visit agent server AG_{i+2} (Figure 4-4 b). Consequently, $shadow_i$ is the new master, monitored by $shadow'_i$.

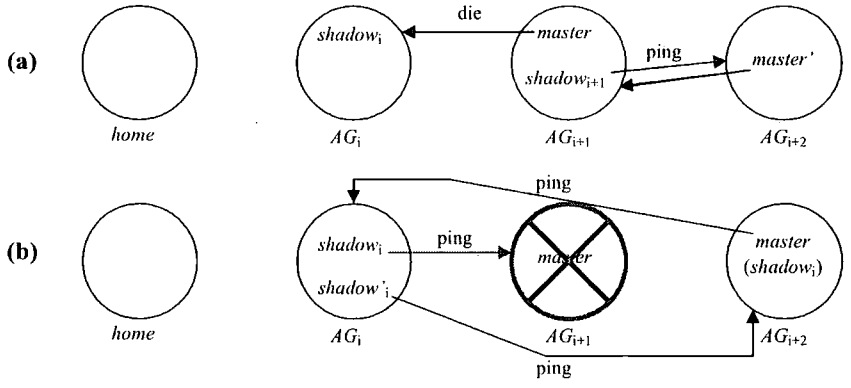


Figure 4-4 Handling a crash at the server occupied by the master

A *master* pings the shadow's agent server AG_{i-1} concurrently with the execution of task *T*. In the normal case (Figure 4-5 a) the master completes its execution and spawns a new clone *shadow'* to monitor the next host in the itinerary AG_{i+1}. Before the master migrates a *die* message is sent to terminate the shadow at AG_{i-1}. If the master detects a shadow crash it raises a local exception to signify the failure of its shadow. The master's exception handler then spawns and dispatches a replacement *shadow''* to the next preceding active agent server, i.e. AG_{i-k} and pinging resumes between the new shadow and the master (Figure 4-5 b). Before the master migrates to the next host it sends a *die* message to terminate the replacement shadow at AG_{i-k}.

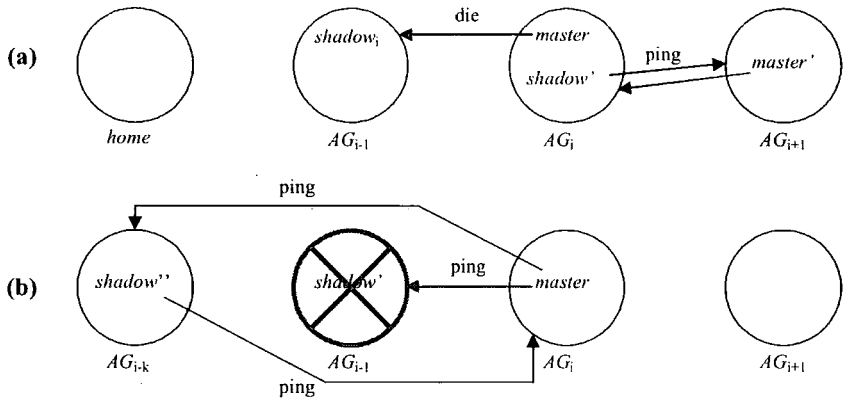


Figure 4-5 Handling a crash at the server occupied by the shadow

The failure model, outlined in section 3, assumes a synchronous network environment and at least once application semantics, i.e. applications must be idempotent. If the mobile shadow scheme is adopted in an asynchronous network environment then false failure suspicions occur due to slow communication links or slow processor speeds. False failure suspicions have the following implications for the mobile shadow scheme in an asynchronous system with at least once and idempotent application semantics:

- Duplicate master instances occur when a shadow falsely suspects a master crash.
- Duplicate shadow instances occur when a master falsely suspects a shadow crash.
- Redundant ping messages. For example, a shadow that is falsely suspected to have crashed by a master results in the replacement shadow and the master sending ping messages. Furthermore, the original shadow continues to ping the agent server occupied by its master.
- Many master instances report back to the home agent server when a shadow falsely suspects a master crash. This increases the load at the home agent server.
- The original master or shadow remains monitoring the agent server occupied by its partner. Consequently a duplicate instance is created if the original master or shadow correctly, or falsely, suspects that the agent server previously occupied by its partner fails by crashing.

It is assumed that an itinerary and mobile agent have the following operations and state:

Itinerary: An itinerary encapsulates a queue, *_destinations*, of agent servers to visit, and a stack, *_visited*, of agent servers visited by the mobile agent.

- *go():* Remove next agent server in the sequence from *_destinations* queue and push onto *_visited* stack. Dispatch mobile agent to the next agent server.
- *skip():* Skip next agent server by removing address from the head of *_destinations* queue.
- *loc = itin.getPrevDestination(k):* Get the address of the k^{th} previous agent server visited.

Mobile Shadow

- *itin:* Itinerary instance.
- *master:* True when the mobile agent is a master.
- *alive:* False when a mobile agent is notified that its shadow or master has crashed.
- *dieProxy:* Reference to shadow that master terminates before migrating to next agent server.
- *shadowProxy:* Reference to master's shadow.
- *masterHost:* Address of master agent server that shadow pings.
- *shadowHost:* Address of shadow agent server that master pings.
- *shadowProxy = spawnShadow():* Spawn a new replica and return its reference. The *dieProxy* is updated to reference the master's previous shadow and *master* is set to *false* in the replica. If the agent is a shadow that has detected its master crash then *masterHost* is set to the next available host in the itinerary. If the agent is a master then *masterHost* is the next host to visit. The *shadowHost* is always the address of the current agent server.
- *PingThread(HostName, proxy):* Thread that pings host *HostName*. The mobile agent *proxy* is notified of a crash by sending it a *pingNotify* message.
- *dispatch(proxy, HostName):* Dispatch mobile agent *proxy* to agent server *HostName*.
- *send(die):* Message that the master sends to terminate its shadow.
- *receive(die):* Shadow listens for die message sent by its master for termination.
- *receive(pingNotify):* Notification of a master or shadow crash.
- *execute():* Start execution at the current agent server.
- *atHome():* Return true if at home agent server.

Figure 4-6 describes the protocol. When the master starts at its home agent server (line 11), i.e. *atHome()=true*, it spawns a shadow (line 13) and migrates to the first host in the itinerary (line 26). If the mobile agent is a master and is at a remote agent server (line 14) it creates a thread to ping the shadow (line 16). Before the master migrates to the next agent server it spawns a new shadow (line 19) and sends a die message (line 20) to terminate the old one. However, if the mobile agent is a shadow, i.e. *master=false*, it invokes *monitorMaster()* (lines 44-53) to create a ping thread to monitor the master's current agent server. Pinging continues if the master is alive and has not dispatched a die message, i.e. *alive=true* and *!receive(die)*.

```
1: master = true
2: alive = true
3:
4: {// application specific task }
5: execute()
6:
7: {// is mobile agent at home agent server }
8: atHome()
9:
10: run(){ // mobile agent execution thread }
11: {if master && atHome()  { // master at home }
12:       { // spawn a shadow }
13:       shadowProxy=spawnShadow()
14: else if master  {// if mobile agent is a master}
15:       { // start thread to ping shadow loc.}
16:       pingShadow( shadowHost )
17:       execute()  { // execute application task }
18:       { // spawn a new shadow }
19:       shadowProxy=spawnShadow()
20:       send(die)  {// terminate previous shadow}
21: else  { // mobile agent is a shadow }
22:       monitorMaster() { // ping master }
23:
24: if master
25:       { // migrate to next agent server }
26:       itin.go()
27: }
```

```
28: pingNotify()  { // callback for ping thread }
29: {  alive=false
30:    if master {// if master then replace shadow}
31:       shadowDispatched=false; k = 2
32:       prev = itin.getPrevDestination( k )
33:       while !shadowDispatched &&  prev != null
34:          try
35:             shadowProxy=spawnShadow()
36:             pingShadow( prev )
37:             dispatch(shadowProxy, prev)
38:             shadowDispatched = true
39:          catch(UnknownHostException)
40:             k++
41:             prev=itin.getPrevDestination( k )
42: }
43:
44: monitorMaster()
45: {  { // start pinging master }
46:    PingThread pinger=new
          PingThread(masterHost, this)
47:    pinger.start()
48:    while(alive && !receive(die))
49:       if  !alive { // if master crash detected }
50:          itin.skip()  { // skip crashed agent server }
51:          shadowProxy=spawnShadow()
52:          master = true  {// change to master status}
53: }
```

Figure 4-6 Mobile shadow scheme pseudocode

If the ping thread detects a crash the *pingNotify()* callback method is invoked (lines 28 - 42) and the alive flag is set to *false* to trigger exception handling activity. If the mobile agent is a master then the shadow exception handler is activated (lines 31 – 41) to spawn a replacement shadow at the first active previous agent server, *itin.getPrevDestination(k)*. The master can then ping the location of the new shadow (line 36). Alternatively, if the mobile agent is a shadow then master exception handling activity is activated (lines 49-52). The master exception handler spawns a new shadow and initialises the shadow to become the new master, i.e. *master = true*.

The mobile shadow exception handling scheme offers the advantage that all agent servers are not revisited in the event of a crash failure, since a replica is available at an agent server that precedes the master. Consequently, there is less information loss. However, greater performance overheads are imposed on a mobile agent since a replica must be spawned by the master before it migrates to the next host in its itinerary. Furthermore, a limited number of remote agent server crashes are addressed.

In this research the following assumptions are made to protect mobile agents from agent server crashes:

- Reliable communication links are assumed.
- All agent servers are correct and trustworthy.
- A mobile agent crashes when its current local agent server halts execution due to a host crash or fault in the agent server process.

- No stable storage mechanism is provided at visited agent servers for the recovery of executing agents.
- *At least once* failure semantics are assumed whereby the agent performs its designated task at least once. If an agent server crashes it is possible to repeat the task at agent servers ignoring those that crashed.
- A mobile agent ignores crashed agent servers.
- A mobile agent visits agent servers to consume information, i.e. agent server state is not modified.
- There are no simultaneous crashes of agent servers where master and shadow operate.

5 Implementation

So far the failure model and design of the mobile shadow scheme for mobile agents to survive agent server crash failures has been discussed. This section describes an IBM Aglets [Oshima98] implementation of the mobile shadow exception handling scheme. Firstly, the rationale for using the IBM Aglets [Oshima98] mobile agent system is outlined in section 5.1. Subsequently, an overview of the implementation is then provided in section 5.2 with the aid of UML class and sequence diagrams.

5.1 Rationale for IBM Aglets

A summary of the features of existing mobile agent systems is provided in chapter 2, section 2.7. Two mobile agent systems were considered for the implementation of the mobile shadow exception handling scheme. These are Ajanta [Tripathi02] and IBM Aglets [Oshima98]. Ajanta [Tripathi02] provides exception handling for mobile agents. IBM Aglets [Oshima98] provides a partial implementation of the OMG MASIF interoperability standard [Milojivcic98]. Neither mobile agent system protects mobile agents from agent server crashes.

Ajanta [Tripathi02], developed at the University of Minnesota, is a mobile agent system for research into exception handling and security. The concept of a guardian software object is used to handle unprecedented exceptions encountered by mobile agents at remote agent servers. An introduction to exception handling in the Ajanta [Tripathi02] mobile agent system is provided in chapter 3, section 4.1. If a mobile agent encounters an unprecedented exception at a remote agent server, it can co-locate with the guardian software object at the home agent server for remedial action. The security model of Ajanta [Tripathi02] focuses on protecting agent server resources from malicious mobile agents. It is assumed that agent servers are not malicious. An Ajanta agent server provides a registry of software services that are available to visiting mobile agents. Each mobile agent is assigned a set of credentials that describe its name, owner, creator and code base. An Ajanta agent server inspects the credentials of a mobile agent to authenticate access to local services. Based upon these credentials an agent server creates a proxy to the

requested software resource. Consequently, mobile agents are not given direct access to software resources. Instead, a proxy controls access to software resources based upon the mobile agent's credentials.

Aglets [Oshima98] is a mobile agent system originally developed by IBM. In August 2000 IBM released Aglets as an open source project at http://aglets.sourceforge.net/. Unlike Ajanta [Tripathi02], IBM Aglets [Oshima98] does not have an explicit model for exception handling. There is also no registry of software services available to visiting mobile agents. Instead, visiting mobile agents can retrieve a list of mobile agents currently executing at the agent server for communication by proxy. In this case mobile agents can interact with resources through static mobile agents. Controlled access to agent server resources is therefore limited with IBM Aglets [Oshima98].

Initially, Ajanta [Tripathi02] was selected as the mobile agent system for the implementation of the mobile shadow exception handling scheme. This was due to the availability of an exception handling model for mobile agents and controlled access to agent server resources. Indeed, an initial experiment using Ajanta [Tripathi02] is detailed in chapter 6 section 2.1. However, there were problems encountered and bugs uncovered. Furthermore, at the time of implementation Ajanta [Tripathi02] was only compatible with Java JDK 1.1. Work was underway, at the University of Minnesota, to migrate Ajanta [Tripathi02] for compatibility with JDK1.2+. However, a release was unavailable at the time of implementation. Consequently, Ajanta [Tripathi02] was abandoned and development migrated to IBM Aglets [Oshima98].

5.2 An IBM Aglets implementation

The IBM Aglets [Oshima98] implementation of the mobile shadow scheme consists of two packages:

- **Shadow package**: Migrates with application mobile agents. Provides the mobile shadow exception handling scheme, including an itinerary pattern and a ping utility.
- **Server package**: Distributed at agent servers visited by the mobile agent. Provides an agent server and a utility that allows agent servers to respond to ping messages from a shadow or master.

This section begins with an outline of the shadow and server packages for the IBM Aglets [Oshima98] implementation of the mobile shadow exception handling scheme. An overview of a mobile agent's life cycle in the mobile shadow exception handling scheme is then provided with the aid of a UML state diagram for the roles of master and shadow. Subsequently,

implementations of the following conceptual operations are outlined with the aid of UML sequence and class diagrams:

- Spawn a shadow.
- Terminate a shadow.
- Dispatch a replacement shadow.
- Ping the availability of an agent server.

Application developers must be able to develop an application task that is independent of the mobile shadow implementation. The section concludes with a description of the classes that enable application development with the mobile shadow exception handling scheme.

5.2.1 Implementation classes

Figure 4-7 illustrates a UML class diagram for the shadow package. The master and shadow mobile agents are both instances of the *MobileShadow* class. This encapsulates the protocol for the mobile shadow exception handler scheme outlined in section 4.

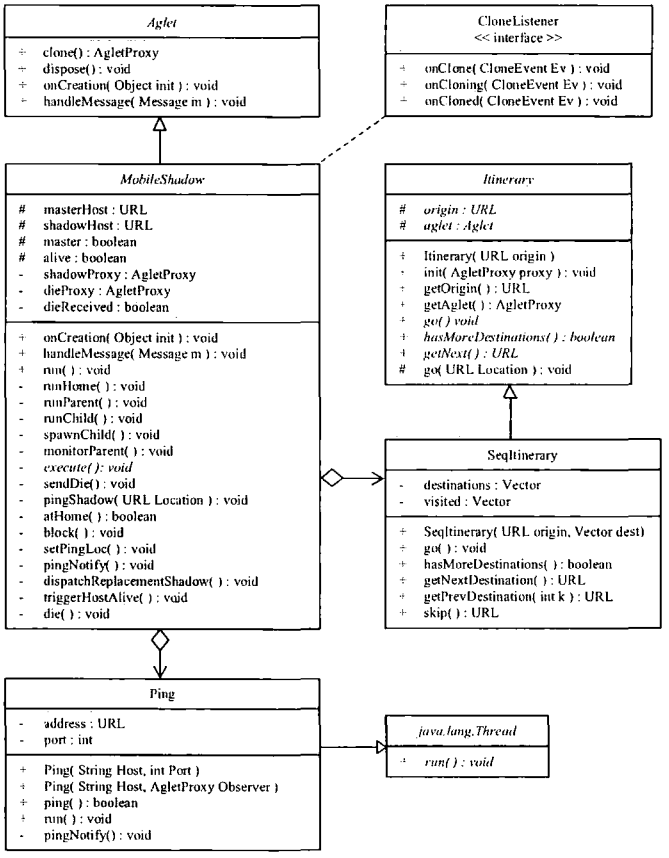


Figure 4-7 UML class diagram for mobile shadow agent package

The *MobileShadow* class contains an instance of a sequential itinerary (class *SeqItinerary*) and a ping utility (class *Ping*) to ping an agent server occupied by a shadow or master. The itinerary is based upon the itinerary pattern of [Aridor98]. Operations are provided to retrieve

the next address from the itinerary, *getNextDestination()*, move the mobile agent to the next agent server, *go()*, and determine if there are more destinations to visit, i.e. *hasMoreDestinations()*. The itinerary pattern outlined in [Aridor98] is extended to log a history of agent servers, *visited*. Each entry in the visited vector represents the URL of a visited agent server. Consequently, it is possible to retrieve the URL of the k^{th} previous agent server, *getPrevDestination(k)*. Another extension to [Aridor98] is the possibility to bypass the next agent server in the itinerary. This is useful when the next agent server has failed by crashing. The *skip()* operation returns the URL of the agent server subsequent to the one bypassed.

A mobile shadow agent pings the availability of its master or shadow using an instance of the *Ping* class. A *Ping* object is transient, i.e. its state is not saved upon migration, and contains the host address and port of the master or shadow's agent server. The *Ping* class executes the *ping()* operation within a thread that continuously monitors the agent server occupied by the master or shadow. Consequently, each agent server assigns security permissions for visiting mobile agents to connect and ping port number 5555.

Figure 4-8 illustrates a UML class diagram for the server package. The *Server* class represents an agent server implementation. The *Server* class is a wrapper for an *AgletsContext* object, i.e. the core IBM Aglet class for sending, receiving and hosting mobile agents. Each agent server has a *propertyFile* field that references the path to a file that configures the agent server.

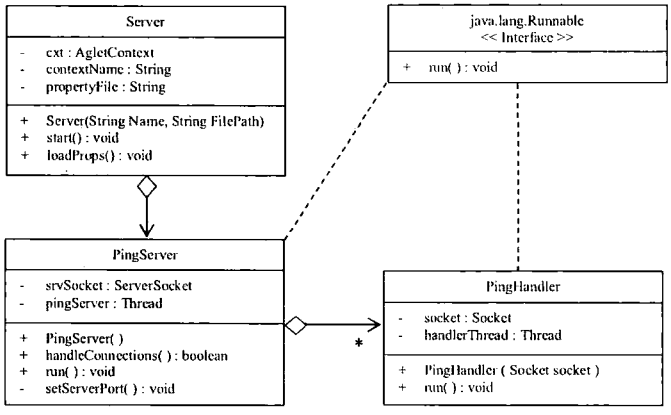


Figure 4-8 UML class diagram for mobile shadow server package

The configuration variables for the server are:

- **AGLET_CLASS_PATH**: Java class path where IBM Aglets [Oshima98] searches for classes referenced by an aglet.
- **AGLET_EXPORT_PATH**: Java class path that represents the code base, i.e. the directory where a remote agent server requests classes.
- **PORT**: The TCP/IP port that the agent server listens to for ping requests.

The *start()* operation boots the agent server. This involves loading configuration variables from file, starting a ping server thread and booting the IBM Aglets [Oshima98] runtime.

Each agent server runs a ping server (class *PingServer*) on a given port. In this implementation port 5555 was selected. Assigning a fixed port number for ping operations is a realistic assumption, since operating systems also provide services at fixed ports. For example, the echo service is assigned to port 7. Class *PingServer* represents a ping server thread that continuously listens for a connection on port 5555. For each request a thread (class *PingHandler*) is spawned that echoes back the string sent.

5.2.2 Mobile shadow life cycle

So far, it has been established that the *MobileShadow* class (Figure 4-7) represents a mobile agent. A master and shadow in the mobile shadow exception handling scheme are both instances of the *MobileShadow* class. Figure 4-9 illustrates a UML state diagram for an instance of the *MobileShadow* class. The *runningMaster* and *runningShadow* superstates describe the execution of the mobile agent as a master and shadow respectively.

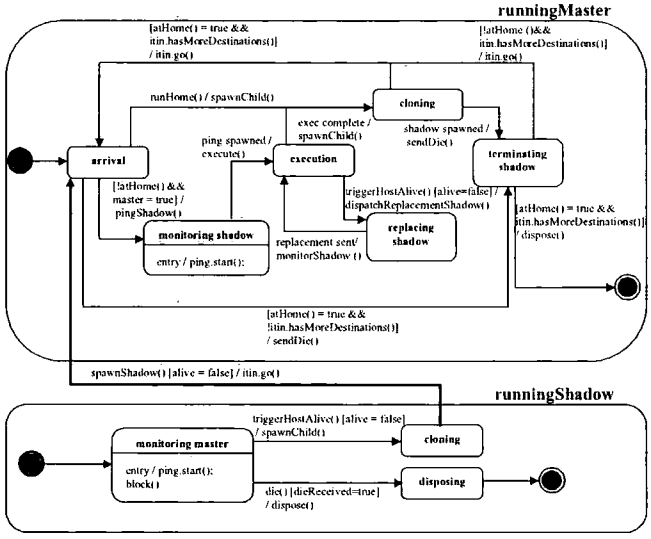


Figure 4-9 UML state diagram for a mobile agent in the mobile shadow scheme

A mobile agent executes the role of a master when it is in the *runningMaster* state. This is synonymous to the *runParent* operation of the *MobileShadow* class (Figure 4-7). The *arrival* state is triggered when a master is created, or arrives, at an agent server in the itinerary. This allows the master to determine the subsequent action for its current location.

When the master is created at the home agent server the *runHome()* operation is executed. This triggers the transition from the *arrival* to the *cloning* state. A shadow is spawned and the master migrates to the next agent server. This activates the transition from the *cloning* to *arrival* state. Consequently, the master is now located at the first remote agent server in the itinerary.

When the master arrives at a remote agent server it invokes the *pingShadow()* operation. This triggers the master to enter the *monitoring shadow* state. Upon entry a ping thread is spawned, *ping.start(),* to monitor the agent server occupied by the shadow. Subsequently, the normal state sequence is triggered, i.e. execute the application task, *execution*, spawn a new shadow, *cloning*, and terminate the old shadow, *terminating shadow*. Figure 4-9 illustrates two transitions from the *terminating shadow* state. The transition to the *arrival* state is triggered when the master has not visited all agent servers in the itinerary, i.e. *itin.hasMoreDestinations() =true*. Alternatively, the master terminates itself when it arrives back at the home agent server, thus completing the itinerary.

When the mobile agent has arrived back at the home agent server and completed its itinerary, it invokes the *sendDie()* operation. This triggers the transition from the *arrival* state to the *terminating shadow* state. Consequently, the master sends a die message, *sendDie()*, to the shadow and terminates itself.

So far, the normal mode of execution for the master has been described. The transition from the *execution* to *replacing shadow* state represents an exception handler for the loss of a shadow due to an agent server crash. The ping thread notifies the master when the agent server occupied by the shadow fails by crashing. When a master receives a notify message from its ping thread it invokes the *pingNotify()* operation. This triggers the master to enter the *replacing shadow* state. The master invokes its *dispatchReplacementShadow()* operation to spawn and dispatch a replacement shadow to the next available agent server. Furthermore, the master invokes the *monitorShadow()* operation to spawn a new ping thread to monitor the agent server occupied by the replacement shadow. The master then resumes the *execution* state.

A mobile agent executes the role of a shadow when it is in the *runningShadow* state. This is synonymous to the *runChild()* operation of the *MobileShadow* class. Initially, a shadow executes the *monitorParent* operation that triggers the *monitoring master* state. In this state the shadow spawns a thread to ping the agent server occupied by its master, *ping.start()*, and then executes the *block()* operation. Subsequently, the shadow waits for receipt of a die message or notification from the ping thread that the agent server occupied by the master has failed by crashing. The *die()* operation is invoked when the shadow receives a die message from its master. This sets the *dieReceived* flag to true and subsequently triggers the transition to the *disposal* state, thus terminating the shadow.

The *triggerHostAlive()* operation is invoked when the shadow receives notification of an agent server crash failure from its ping thread This sets the *alive* flag to false, thus triggering the transition to the *cloning* state. This transition represents an exception handler for the loss of

a master, due to an agent server crash. Subsequently, the shadow spawns a new shadow and migrates to the next agent server in the itinerary. This is denoted by the transition from the *cloning* state, in the *runningShadow* super state, to the *arrival* state of the *runningMaster* super state. The shadow is now the new master.

5.2.3 Spawning a shadow

A mobile agent, in the mobile shadow exception handling scheme, spawns a shadow by invoking the *spawnChild* operation (see *MobileShadow* class, Figure 4-7). The following requirements must be taken into account when a shadow is spawned. Firstly, the master must be able to produce a replica, or shadow, with an initial state. Secondly, both a shadow and master must be aware of the location of its partner. This enables a shadow and master to ping the availability of the agent server occupied by its partner. Finally, the master must maintain a communication reference to the current shadow before a new shadow is spawned to monitor the next agent server in the itinerary. Consequently, the master must be able to instruct the current shadow to dispose of itself.

IBM Aglets [Oshima98] provides the ability to produce a replica of the state and behaviour of a mobile agent in volatile memory. The IBM Aglets specification [Oshima98] describes this as a clone operation that is synonymous to spawning a shadow in the mobile shadow scheme. The *Aglet* class (Figure 4-7) provides the *clone* operation to produce a replica mobile agent. When a clone operation is performed, a proxy to the new clone is created. Consequently, the proxy can be used to dispatch the clone.

Most Java mobile agent systems allow reactions to be developed for a mobile agent in response to mobility events, e.g. dispatch and arrival at an agent server. IBM Aglets [Oshima98] allows a mobile agent to react when a clone operation is performed. This is achieved using call back operations. A mobile agent implements the *CloneListener* interface (Figure 4-7) and registers for notification of clone operations, i.e. *addCloneListener(this)*. Table 4-3 lists the clone call back operations, when they are invoked and on which object, i.e. the master or clone.

operation	when invoked	object	function
onCloning	before clone operation	master	prepare state for cloning
onClone	when clone is created	clone	initialise unique state in clone
onCloned	after clone created	master	clean up state of master

Table 4-3 Aglet clone operations

Figure 4-10 illustrates the sequence of operations that occur when a master spawns a shadow. Before a new shadow is spawned a proxy to the current shadow is saved, *dieProxy = shadowProxy*, so that the master can terminate it before migrating to the next agent server in the itinerary. The master invokes the *spawnChild* operation to spawn a shadow. The *clone* operation is then performed and a reference to the new shadow is saved, *shadowProxy*. The *onClone()* is then invoked on the shadow which sets the shadow to a default state, i.e. it is not a master (*master=false)* and the agent server occupied by the master is currently alive (*alive=true)*. The itinerary of the shadow is then initialised with the shadow's proxy. This is necessary so that the itinerary operates on the new shadow, not the master. Finally, the *onCloned()* operation is invoked when cloning is complete.

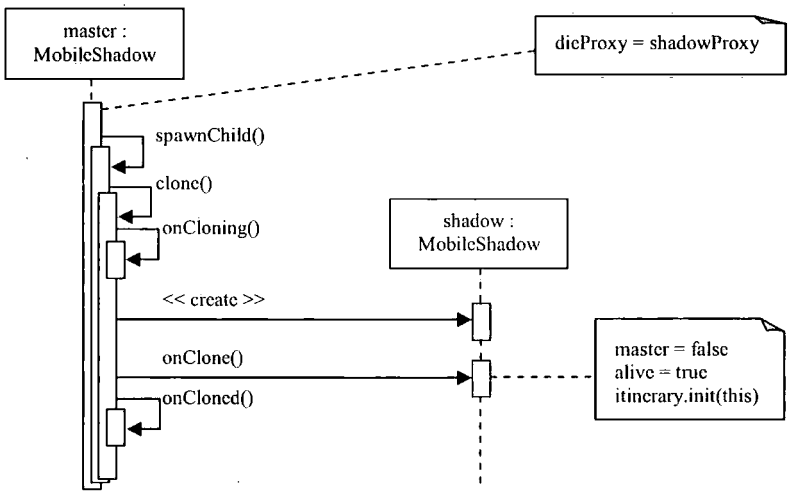


Figure 4-10 UML sequence diagram for spawning a shadow

With respect to the mobile shadow exception handler scheme, a shadow must ping the location of its master. Similarly, a master must ping the location of its shadow. Recall that the master and shadow are both instances of the *MobileShadow* class (Figure 4-7). The *masterHost* and *shadowHost* variables log the URL for the agent server occupied by the master and shadow respectively. The master initialises these variables in the *onCloning* operation.

Figure 4-11 illustrates the behaviour of a clone operation performed by the master. There are three scenarios in which a shadow is spawned:

- **Migration of master to next agent server in the itinerary**: Before the master migrates to the next agent server in the itinerary a new shadow is spawned. The *masterHost* variable is configured as the URL of the next agent server in the itinerary.
- **Shadow replaces its master lost due to an agent server crash**: The *masterHost* variable is configured as the URL of the next available agent server in the itinerary.
- **Master replaces a shadow lost due to an agent server crash**: The *masterHost* variable is configured as the URL of the agent server currently occupied by the master.

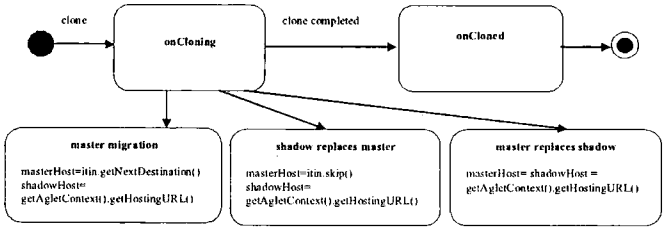


Figure 4-11 UML state diagram for aglets clone operation

In each scenario the *shadowHost* variable is initialised with the URL of the current agent server. Consequently, when the master arrives at the next agent server in the itinerary the *shadowHost* variable refers to the URL of the agent server occupied by the shadow.

5.2.4 Terminating a shadow

A master employs synchronous message passing to terminate a shadow. An aglets message is an object. For example, *new Message("die")* creates a message of type *die*. The syntax for sending a synchronous message is *receiver.sendMessage(Message)* where *receiver* is a proxy to the receiving mobile agent. The sender blocks until the receiver has handled the message.

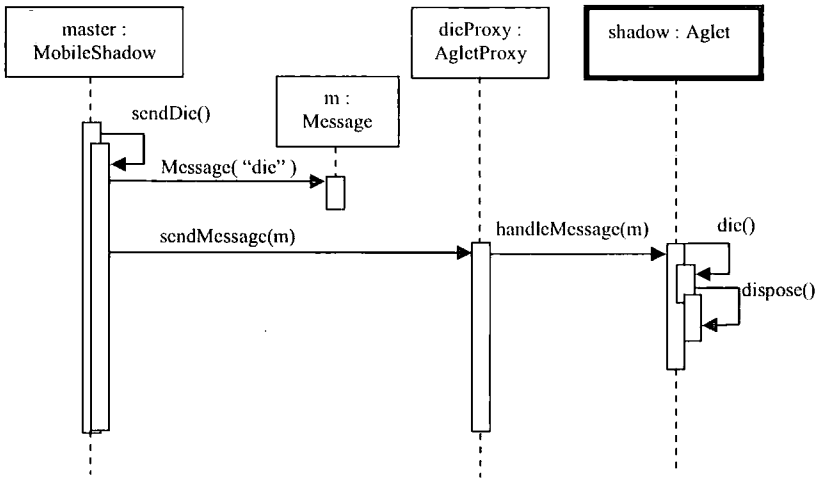


Figure 4-12 UML sequence diagram for master terminating a shadow

Figure 4-12 illustrates a UML sequence diagram for the *sendDie* operation, invoked by the master to terminate its shadow. A die message is created by the master, *Message("die")*, and dispatched to the shadow via its proxy, *dieProxy.sendMessage(m)*. Whenever an IBM Aglets [Oshima98] mobile agent receives a message the *handleMessage* operation is invoked. This operation allows the developer to react to different messages. If a *die* message is received, *msg.sameKind("die")=true*, the shadow disposes itself, *dispose()*, and sets its *dieReceived* flag to true. The change in state of the *dieReceived* flag gracefully exits the run thread of the shadow. Consequently, the shadow is terminated.

5.2.5 Dispatching a replacement shadow

A master monitors the agent server occupied by its shadow. If the master detects the failure of its shadow's agent server, it spawns a replacement and dispatches it to the next available agent server previously visited. The procedure for sending a replacement shadow (Figure 4-13) loops until the shadow is successfully dispatched or no available agent server is found, i.e. *[prevDestination != null && !dispatched]*.

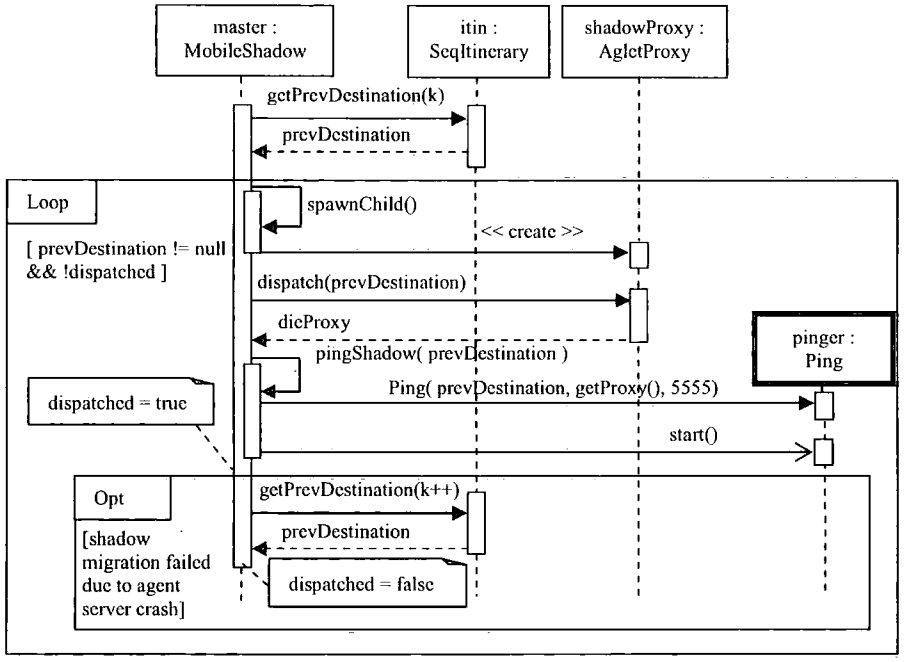


Figure 4-13 UML sequence diagram for dispatching a replacement shadow

Before entering the loop the master uses the *SeqItinerary* class (Figure 4-7) to determine the URL of the last agent server visited prior to that currently occupied by the shadow. This is achieved by the *getPrevDestination* operation. The loop begins by spawning a shadow and initialising its proxy, *shadowProxy*. The shadow is dispatched to the previously visited agent server via its proxy, *dispatch(prevDestination)*. Furthermore, the *dieProxy* is updated to reference the replacement shadow so that it can be terminated before the master migrates to the next agent server in the itinerary. Finally, the master spawns a thread to ping the agent server occupied by the replacement shadow, *pingShadow(prevDestination)* and sets the dispatched flag to *true* to exit the loop.

The dispatch method raises an exception if the agent server referenced by *prevDestination* has failed by crashing. Consequently, the URL of the next available previous agent server visited in the itinerary is retrieved. The dispatched flag is reset to *false* and the loop continues. The option box in Figure 4-13 highlights the sequence of events for this scenario.

5.2.6 Pinging an agent server

The master and shadow spawn a ping thread (class *Ping* Figure 4-7) to monitor the availability of the agent server occupied by its partner. Figure 4-14 illustrates the creation of a ping thread and the execution sequence for notification of an agent server crash failure.

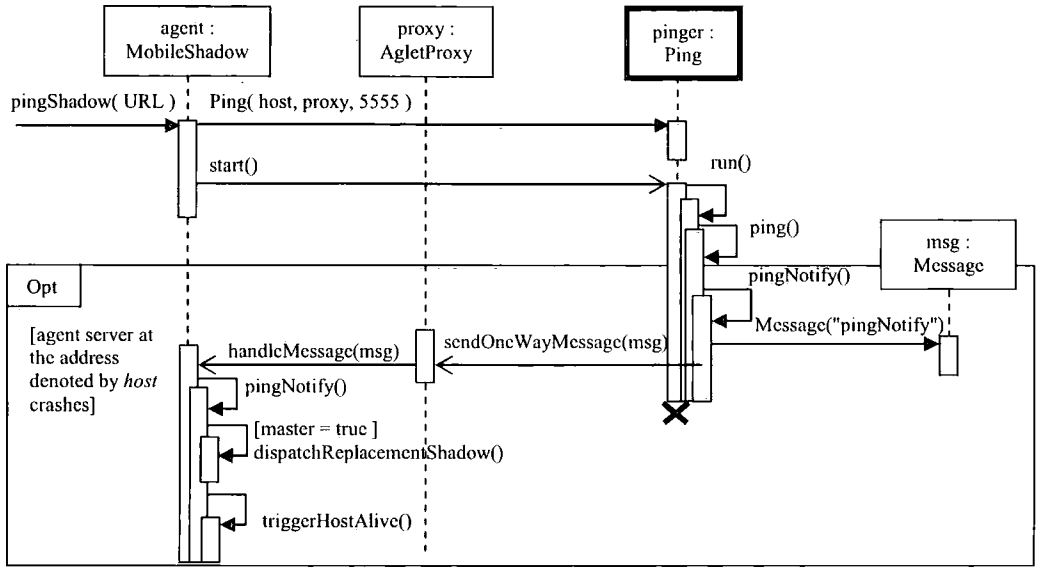


Figure 4-14 UML sequence diagram for notification of an agent server crash failure

A ping thread (class *Ping*) accepts three parameters for creation:

- **Host:** The URL string representing the agent server to ping.
- **Proxy:** A proxy to the mobile agent. This is used to notify the mobile agent if an agent server crash is detected.
- **Port number:** All agent servers listen for ping messages on port 5555.

When the thread is started it invokes its *ping* method. The ping method connects to a Java TCP/IP server socket on port 5555 at the host denoted by the agent server URL. A string is then sent and echoed back by an instance of the *PingHandler* class at the agent server. The ping method continuously pings while the agent server is alive. If the ping utility fails to connect to a Java server socket, it invokes its *pingNotify* operation. The option box in Figure 4-14 highlights the subsequent sequence of events. The ping utility notifies a mobile agent, via its proxy, that the agent server has failed by crashing. When a *MobileShadow* instance receives a *pingNotify* message it invokes its own *pingNotify* operation. If the mobile agent is a master, *master = true,* a replacement shadow is dispatched, *dispatchReplacementShadow()*. Furthermore, the alive flag is set to false, *triggerHostAlive()*. If the mobile agent is a shadow then the state change of the alive flag triggers replacement of the master.

5.2.7 Application development

So far, it has been established that the master, is responsible for performing an application specific task at each agent server visited in the itinerary. The mobile shadow exception handling scheme is intended to be transparent to application developers. This implies the following requirements for application development. Firstly, it must be possible to develop an application specific task that is performed by the master at each agent server in the itinerary. Furthermore, an application task may require access to local software resources at the current agent server. For example, a database resource may be queried or an application object may perform a task on behalf of the mobile agent. It is assumed that a mobile agent does not modify the local resources at an agent server. Subsequently, prior to execution, the developer must be able to locate local resources at each agent server.

The UML class diagram in Figure 4-15 illustrates the abstract classes for application development in the mobile shadow exception handling scheme. So far, it has been established that the master and shadow are both instances of the *MobileShadow* class. Furthermore, the *MobileShadow* class is initialised with an instance of the *ApplicationObject* class and a vector containing the URLs of agent servers to visit in the itinerary. These are encapsulated within an *Args* object. The application developer must develop a class that derives from the *ApplicationObject* class. This class represents the application specific task performed at each agent server and contains two operations that must be implemented, *execute* and *initResources*. The *execute* operation is invoked by the master to perform the application specific task at each agent server. The *initResources* operation uses an instance of the *ResourceManager* class to locate application specific resources at the agent server. Figure 4-15 illustrates that a mobile agent, *MobileShadow*, is assigned the role of resource manager.

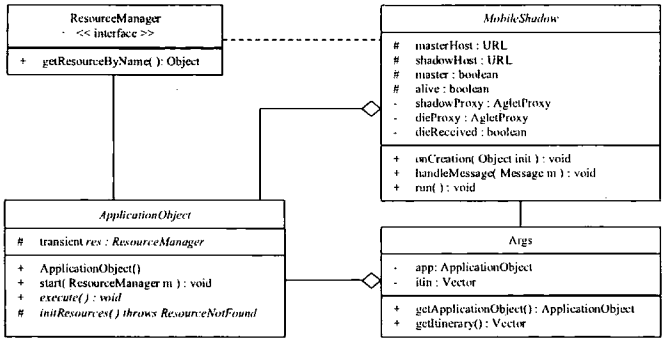


Figure 4-15 UML class diagram for application development in the mobile shadow scheme

The IBM Aglets [Oshima98] implementation of the mobile shadow exception handling scheme provides the following concept for mobile agents to locate resources at an agent server. An agent server, AG_k, hosts a stationary mobile agent, SM_k, for each resource, R_k. This acts as an intermediary, or proxy, between a visiting mobile agent and the resource. It is assumed that

the mobile agent system allows visiting mobile agents to inspect a list, $execution_{list}$, of active mobile agents at the agent server. A mobile agent can be retrieved from the list identified by its unique class name. A visiting mobile agent, MA_k, locates a resource, R_k, indirectly by retrieving a handle to the stationary mobile agent, SM_k, from the list of active mobile agents, $execution_{list}$, at the agent server. Subsequently, the mobile agent communicates with SM_k to use resource R_k. IBM Aglets [Oshima98] provides the *AgletContext* class for visiting mobile agents to query the environment of its current agent server. This provides the *getAgletProxies* operation to allow mobile agents to retrieve a list of executing mobile agents. Each list entry is a proxy. Subsequently, the *getResourceByName* operation retrieves a proxy to a stationary mobile agent at the agent server. The application developer forwards requests to the stationary mobile agent to use the resource.

The UML sequence diagram in Figure 4-16 illustrates the scenario for a master executing an application task. When a master arrives at the next agent server in the itinerary it invokes the *execute* operation. This starts the application task by invoking the start operation, *start(master)*, of the application object. The start operation accepts an instance of the master as the resource manager. When an application task is started the *initResources* operation is invoked. This is an abstract operation that is overridden by the application developer to access local resources at the agent server. A resource is accessed using the *getResourceByName* operation of the resource manager. Application developers specify the full class name of the stationary mobile agent that acts as a proxy to the required resource, e.g. *shadow.DBManager*. Once local resources have been located the application task is executed at the agent server, i.e. *execute*.

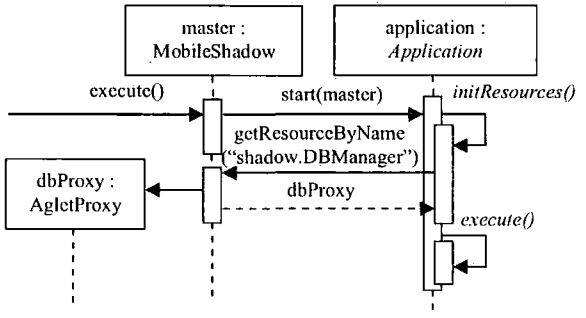


Figure 4-16 UML sequence diagram for executing an application task

6 Demonstration

This section concludes with a demonstration of the IBM Aglets [Oshima98] implementation of the mobile shadow exception handling scheme. The demonstration is deployed within a 10mbps local area network using four 64MB Intel Pentium II 400Mhz (Celeron) PC's running RedHat Linux 7.2 and IBM Aglets version 2.0.2. The case study scenario, outlined in chapter 5, is used, i.e. a single mobile agent visits three agent servers to determine the best buy for a specific product. The itinerary is illustrated in Figure 4-17.

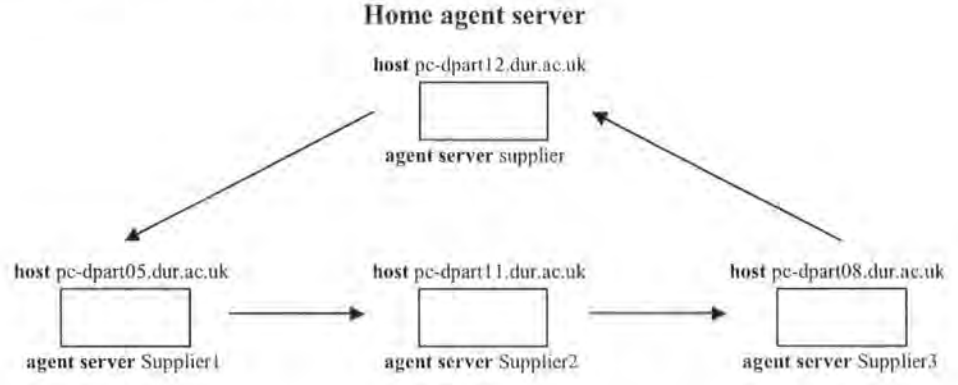


Figure 4-17 Itinerary

Figure 4-18 illustrates a sample agent server running at host *pc-dpart08.dur.ac.uk*. When an agent server is started its environment is configured. Configuration tasks include: starting a thread to accept ping requests and initializing the IBM Aglets [Oshima98] runtime. When the agent server is successfully configured it displays its name and waits for visiting mobile agents. Configuration of the IBM Aglets [Oshima98] agent server is described in section 5.1.

Figure 4-18 Agent server running at host pc-dpart08.dur.ac.uk

The execution of a mobile agent, with no agent server crash failures encountered, is illustrated in Figure 4-19. Terminal windows are used to display the output for each agent server visited in the itinerary. A mobile agent is dispatched to visit three agent servers. At each agent server the following activities are performed by the master:

- Execute the application task.
- Spawn a shadow to ping the availability of the next agent server in the itinerary.
- Terminate the old shadow, at the preceding agent server, before migrating to the next agent server in the itinerary.

Figure 4-19 Normal mobile agent execution

For example, when the master has completed execution at the last agent server in the itinerary, *atp://pc-dpart08.dur.ac.uk*, the following activities occur:

- Spawn a new shadow to monitor the availability of the home agent server, *atp://pc-dpart12.dur.ac.uk/supplier*.
- Terminate the shadow at agent server *atp://pc-dpart11.dur.ac.uk/Supplier2*.
- Migrate to the home agent server, *atp://pc-dpart12.dur.ac.uk/supplier*.

Recall that the mobile shadow exception handler scheme contains two exception handlers. The master exception handler is invoked within the shadow when it detects the crash of the agent server occupied by its master. The shadow exception handler is invoked within the master when it detects the crash of the agent server occupied by its shadow. Section 6.1 provides a demonstration for the master exception handler. Finally, section 6.2 concludes with a demonstration of the shadow exception handler.

6.1 Master exception handler

The following simulation triggers the master exception handler. When the master arrives at host *pc-dpart05.dur.ac.uk* it terminates the agent server, using the Java command *System.exit(1)*. Consequently, the shadow must detect the agent server crash and replace the master by migrating to the next available agent server in the itinerary, i.e. *atp://pc-dpart11.dur.ac.uk/Supplier2*.

Figure 4-20 illustrates a terminal window for each agent server visited in the itinerary. The output for host *pc-dpart12.dur.ac.uk* illustrates that the master is initially dispatched to the first agent server in the itinerary, i.e. *atp://pc-dpart05.dur.ac.uk/Supplier1*. Before the master migrates, it spawns a shadow to monitor the availability of agent server *atp://pc-dpart05.dur.ac.uk/Supplier1*. Subsequently, the shadow detects the crash and activates the master exception handler. This spawns a new shadow, with identifier 0e006e39b6160df1, and then dispatches the current shadow to the next agent server, *atp://pc-dpart11.dur.ac.uk/Supplier2*. At this point the shadow has replaced the master and normal execution resumes. The replacement master completes the itinerary, at agent servers *atp://pc-dpart11.dur.ac.uk/Supplier2* and *atp://pc-dpart08.dur.ac.uk/Supplier3*, before reporting back to the home agent server at *atp://pc-dpart12.dur.ac.uk/supplier*.

```
CONFIGURING AGENT SERVER...
AGENT SERVER : supplier
**********************************************

BUILDING DOM FROM XML FILE orderFormat.xml
DISPATCHING AGENTS FOR EACH PRODUCT
SIMULATING CRASH AT AGENT SERVER NUMBER :: 1
MASTER STARTING ITINERARY
MASTER MIGRATING TO atp://pc-dpart05.dur.ac.uk/Supplier1

SHADOW 6f6562caad71b530 - MONITORING MASTER AT @ atp://pc-dpart05.dur.ac.uk/Supplier1

AGENT SERVER CRASH DETECTED
SHADOW DETECTED MASTER FAILURE - MASTER EXCEPTION HANDLER ACTIVATED
SHADOW REPLACING MASTER

SHADOW MIGRATING TO atp://pc-dpart11.dur.ac.uk/Supplier2
SHADOW 0e006e39b6160df1 - MONITORING MASTER AT @ atp://pc-dpart11.dur.ac.uk/Supplier2

SHADOW 0e006e39b6160df1 TERMINATED BY DIE MESSAGE
ITINERARY COMPLETE
MASTER TERMINATING SHADOW @ atp://pc-dpart08.dur.ac.uk/Supplier3
```

```
CONFIGURING AGENT SERVER...
AGENT SERVER : Supplier2
**********************************************

MASTER EXECUTING APPLICATION TASK
MASTER COMPLETED EXECUTION - SPAWNING SHADOW
MASTER TERMINATING SHADOW @ atp://pc-dpart12.dur.ac.uk/supplier

SHADOW 592867d1ac542088 - MONITORING MASTER AT @ atp://pc-dpart08.dur.ac.uk/Supplier3

MASTER MIGRATING TO atp://pc-dpart08.dur.ac.uk/Supplier3
SHADOW 592867d1ac542088 TERMINATED BY DIE MESSAGE
```

```
CONFIGURING AGENT SERVER...
AGENT SERVER : Supplier3
**********************************************

MASTER EXECUTING APPLICATION TASK
MASTER COMPLETED EXECUTION - SPAWNING SHADOW

MASTER TERMINATING SHADOW @ atp://pc-dpart11.dur.ac.uk/Supplier2
SHADOW 4fbad4a05c7bf46d - MONITORING MASTER AT @ atp://pc-dpart12.dur.ac.uk/supplier
MASTER MIGRATING TO atp://pc-dpart12.dur.ac.uk/supplier
SHADOW 4fbad4a05c7bf46d TERMINATED BY DIE MESSAGE
```

Figure 4-20 Master exception handler

6.2 Shadow exception handler

The following scenario is simulated to trigger the shadow exception handler with respect to the itinerary described in Figure 4-17. When the master arrives at agent server *atp://pc-dpart11.dur.ac.uk/Supplier2* the agent server at host *pc-dpart05.dur.ac.uk* is manually terminated. Consequently, the master should detect that the agent server occupied by its shadow has crashed. The shadow exception handler must be activated to dispatch a replacement shadow to the next available preceding agent server, *pc-dpart12.dur.ac.uk/supplier*. When the master has completed execution at agent server *pc-dpart11.dur.ac.uk/Supplier2* it should spawn a new shadow and then terminate the replacement shadow at *pc-dpart12.dur.ac.uk/supplier*.

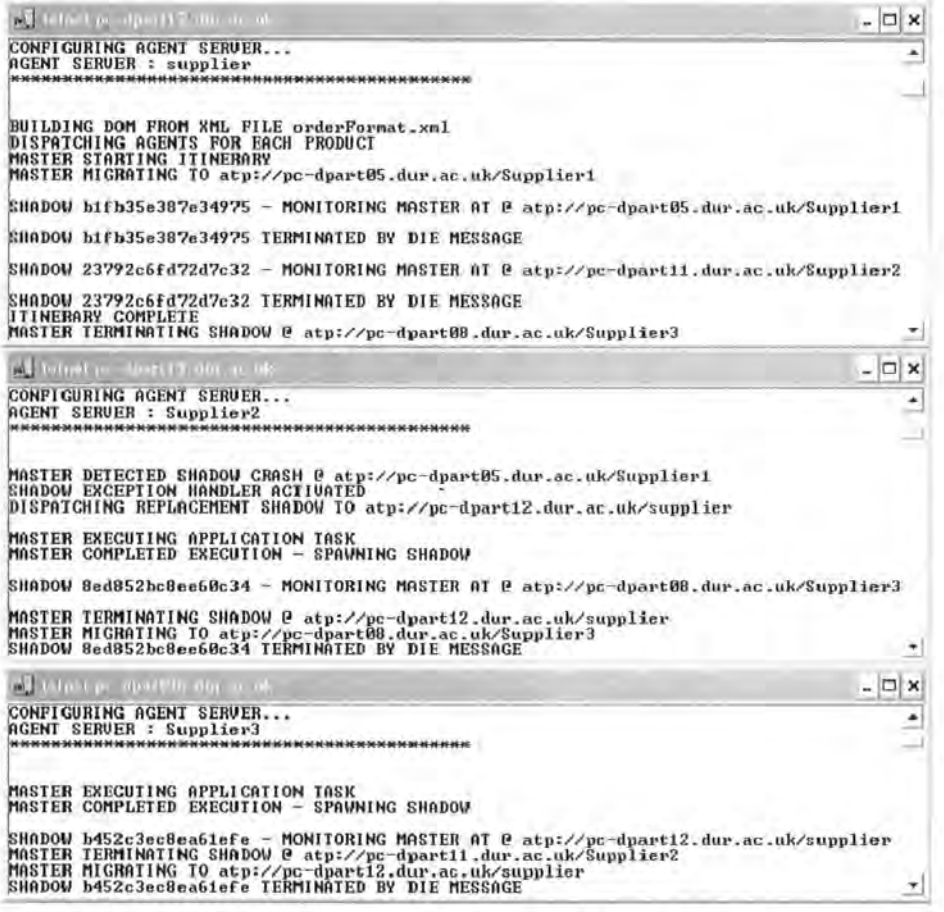


Figure 4-21 Shadow exception handler

Figure 4-21 illustrates a terminal window for each agent server visited in the itinerary. The output for host *pc-dpart12.dur.ac.uk* illustrates that the master is initially dispatched to the first agent server in the itinerary, i.e. *atp://pc-dpart05.dur.ac.uk/Supplier1*. When the master has completed its execution at *atp://pc-dpart05.dur.ac.uk/Supplier1* it spawns a shadow to monitor the availability of agent server *atp://pc-dpart11.dur.ac.uk/Supplier2*. The master then migrates to the next agent server in the itinerary, *atp://pc-dpart11.dur.ac.uk/Supplier2*. At this point the agent server at host *pc-dpart05.dur.ac.uk* is manually terminated. The output for the agent server at host *pc-dpart11.dur.ac.uk* illustrates that the crash is detected and the shadow

exception handler is activated. Subsequently, a replacement shadow is dispatched to the next available preceding agent server, *atp://pc-dpart12.dur.ac.uk/supplier*, and the master continues with its execution. Subsequently, the output for host *pc-dpart-12.dur.ac.uk* illustrates that the replacement shadow monitors its master at agent server *atp://pc-dpart11.dur.ac.uk/Supplier2* and is terminated when the master completes its execution.

7 Summary

This chapter has outlined the design and implementation of the mobile shadow exception handling scheme. The mobile shadow exception handling scheme provides a fault tolerant service for maintaining mobile agent availability in the presence of agent server and host crashes. This service is embedded within the application mobile agent. Several important design issues were considered.

Firstly, existing research into mobile agents surviving agent server and host crashes was considered. Most of the existing fault tolerant mobile agent systems use spatial or temporal replication. Spatial replication defines a set of agent servers for each stage in the itinerary. Each agent server provides an equivalent service and hosts a replica mobile agent. Temporal replication advocates one or more visited agent servers monitoring the current agent server. If an agent server crash occurs, a mobile agent is dispatched to the next agent server in the itinerary or an alternative agent server. Temporal replication was selected to minimise additional communication overheads imposed by fault tolerance. This is important to preserve the potential savings in bandwidth offered by the mobile agent paradigm.

Secondly, an exception handling model for mobile agents was presented to describe how mobile agents interact with software services at remote agent servers. Furthermore, the control flow for raising and signalling exceptions was defined. In the event that an agent server or host crash occurs exception handling is required to perform application specific action so that the mobile agent can complete the itinerary.

Thirdly, a failure model for mobile agent systems has been outlined. Chapter 3 highlighted the need for a failure model for mobile agent systems. This chapter has focused specifically on crash failures of mobile agent systems and presented a failure model suitable for applications that are idempotent, e.g. information retrieval or network monitoring. A failure model is required for any fault tolerant system to define the ways in which failures occur and the assumptions made concerning the system environment.

The following chapter introduces the design and implementation of an experimental case study environment for the mobile shadow exception handler scheme.

Chapter 5 A Case Study Application

1 Introduction

This research employs a Java case study application to provide an experimental environment for simulation of agent server crash failures and subsequent analysis of the mobile shadow exception handling design. A frequently adopted application domain for mobile agents is within an electronic commerce business supply chain [Vogler97, Dasgupta99].

In [Pears03, Pears03b] a case study environment for the mobile shadow exception handling scheme is outlined and the overheads introduced on the trip time in the event of an agent server crash are investigated. This chapter describes the requirements and architecture for the case study environment. The application case study provides an experimental environment for an implementation of the mobile shadow scheme introduced in chapter 4. The case study environment provides the ability to simulate agent server crash failures to exercise the mobile shadow crash exception handler.

This chapter outlines the case study architecture of [Pears03, Pears03b]. There are two implementations of the case study architecture. Firstly, an Ajanta [Tripathi02] implementation is described. The Ajanta [Tripathi02] implementation of the case study architecture is used to compare the performance of the mobile shadow exception handling scheme against an exception handler that uses a timeout mechanism. In chapter 4, section 5.1, it was mentioned that implementation anomalies were encountered with the Ajanta mobile agent system [Tripathi02]. Furthermore, a random agent server crash could not be simulated. Consequently, an IBM Aglets [Oshima98] implementation was performed. The IBM Aglets [Oshima98] implementation of the case study architecture allows the overheads of the mobile shadow exception handling scheme to be investigated in the event of a random agent server crash. Experiments conducted using the case study architecture are outlined in chapter 6, section 2.

2 Application case study architecture

Figure 5-1 illustrates the case study architecture. The supply chain case study scenario executes within a local area network, each node hosting an agent server that represents a supplier of computer hardware components. The case study application enables a supplier to replenish stock using mobile agent technology. A mobile agent is dispatched, for each product component, to several known suppliers to dynamically determine the best buy with respect to delivery date and price. The product components, order constraints and a list of potential supplier agent servers are outlined in an XML file.

Figure 5-1 Supply chain case study architecture

Figure 5-2 illustrates an example XML file for the order of a hard drive component. An order for stock replenishment is assigned a total budget and delivery deadline. Each order contains a list of product items that require stock replenishment. A product item in the XML file includes: the class of product (e.g. hard drive), required stock, delivery date, a list of known suppliers and search criteria parameters. Consequently, the supplier that provides the cheapest product and earliest delivery date offers the best buy. Figure 5-2 illustrates an order for stock replenishment for 25 IDE hard drives with a capacity of 30 GB to be received by 1/5/2004. Three known suppliers are listed.

```
<?xml version="1.0"?>
<ORDER budget="100.00" deadline="1/5/2004">
   <ITEM>
      <CLASS>Hard Drive</CLASS>
      <STOCK>25</STOCK>
      <DELIVERY>1/5/2004</DELIVERY>
      <SUPPLIER>
            <URN>atp://pc-dpart08.dur.ac.uk:4434</URN>
      </SUPPLIER>
      <SUPPLIER>
            <URN>atp://pc-dpart05.dur.ac.uk:4434</URN>
      </SUPPLIER>
      <SUPPLIER>
            <URN>atp://pc-dpart11.dur.ac.uk:4434</URN>
      </SUPPLIER>
      <PARAMETER>
            <NAME>interface</NAME>
            <VALUE>ide</VALUE>
      </PARAMETER>
      <PARAMETER>
            <NAME>size</NAME>
            <VALUE>30</VALUE>
      </PARAMETER>
   </ITEM>
</ORDER>
```

Figure 5-2 Xml order criteria

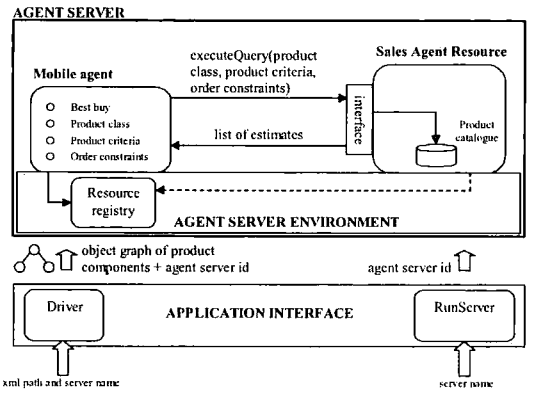


Figure 5-3 Supplier host architecture

Figure 5-3 illustrates the architecture of a supplier host in the case study. Each supplier runs an agent server and application interface. It is assumed that suppliers host the same agent server platform, e.g. IBM Aglets [Oshima98], for interoperability. Furthermore, access to the product catalogue is assumed to be in the same business domain. Consequently, the case study scenario is applicable to small businesses that replenish stock from a manufacturer or mail order company.

A *sales agent* resource enables visiting mobile agents, from trusted suppliers, to query the supplier's product catalogue for item prices and availability. The sales agent is a software object located at each agent server that provides a standard interface for queries to the product catalogue. Each supplier provides a sales agent to implement a standard set of queries on the product catalogue for a given business domain. For example, all hard drive products may be queried by capacity and interface type (SCSI, FLASH or IDE). This requires the developer of visiting mobile agents to be aware of the query parameters for each product in the business domain. A visiting mobile agent invokes the *executeQuery* operation of the sales agent interface that accepts the product class, product criteria and order constraints as parameters. The sales agent then queries the product catalogue for the given class of product. A list of estimates that match the product criteria and order constraints is then returned to the visiting mobile agent.

The *product catalogue* stores information for products sold by the supplier including: item id, manufacturer id, stock and price. The supplier is free to choose the structure and format for the product catalogue, e.g. XML or a database may be used. This is possible since the sales agent provides a standard interface to query the information stored in the product catalogue for a given business domain. Mobile agents can query product catalogues that are of different formats and structure. However, there is the restriction that each supplier must comply with the interface provided by the sales agent.

The *application interface* contains two utility classes, *Driver* and *RunServer*, to boot a supplier agent server. The *Driver* utility starts an instance of a supplier agent server that

replenishes stock for product items listed in an XML file. Two parameters are accepted: the path to the XML file that describes the product components for stock replenishment and a name for the supplier agent server. The *Driver* utility parses the XML file to create an object graph of product components that includes: order constraints, product criteria and a list of suppliers. Subsequently, an agent server is then booted with the object graph. The agent server traverses the object graph and dispatches a mobile agent for each product item. Finally, the *RunServer* utility starts an instance of a supplier agent server identified by a given name.

The case study application assumes that the product catalogue is represented by a database at each supplier. To facilitate the configuration of the experiment environment each supplier is expected to represent the product catalogue using a mysql database. Sections 2.1 and 2.2 describe the implementation classes for the sales agent and application interface.

2.1 The sales agent

Figure 5-4 illustrates a UML class diagram for the implementation of the sales agent resource. The *Estimate* and *SalesAgent* classes are distributed at each agent server:

- **Estimate**: Represents a matching product offered by the supplier. A matching product is described by item id, total price and delivery date.
- **SalesAgent**: Implementation of the sales agent resource that provides an interface for querying the product catalogue database. Visiting mobile agents invoke the *executeQuery* operation to forward a hashtable of product criteria, the product class and order constraints.

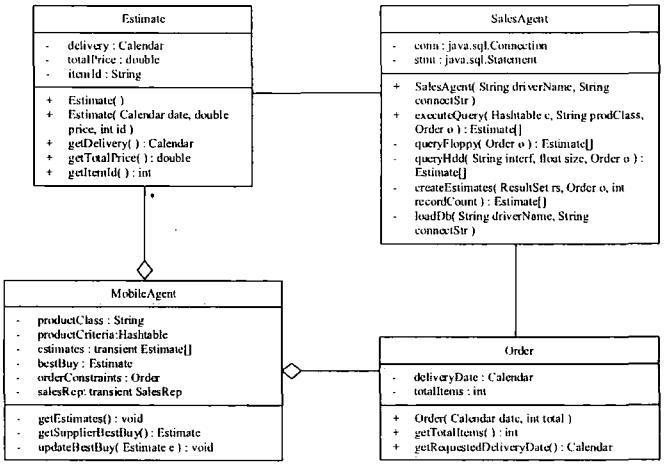


Figure 5-4 UML class diagram for sales agent implementation

A mobile agent in the case study (see *MobileAgent* class[2] Figure 5-4) filters the best buy from an itinerary of suppliers for a given product component described by:

- **Product class**: A string that represents the class of product, e.g. hard drive.

[2] The API for most mobile agent systems enforces the developer to inherit from an abstract class that represents a mobile agent. The class diagram in Figure 5-4 is intended to be independent of mobile agent systems. Consequently, common state and operations are included in the mobile agent class.

- **Product criteria**: A hash table containing search criteria attributes and values for the product, e.g. hard drives, are queried based upon interface type and capacity attributes.
- **Order constraints**: Product order constraints including delivery date and total stock.

When the mobile agent has returned to the home agent server the *bestBuy* attribute contains an estimate from the supplier that offers the best buy. The supplier that provides the best buy offers the cheapest price and earliest delivery date.

The UML sequence diagram in Figure 5-5 illustrates the interaction between an application mobile agent and a sales agent to retrieve a list of matching estimates. When a mobile agent arrives at a supplier agent server it locates the sales agent. Most mobile agent systems provide a registry of resources, at each agent server, that mobile agents can use to access a proxy to a software resource. The *executeQuery* operation is then invoked on the sales agent with the product class, product criteria and order constraints as parameters. The sales agent examines the product class parameter to determine which query to perform on the product catalogue database. Subsequently, the product criteria names and values are retrieved from the product criteria hashtable and forwarded to the appropriate query operation. In the scenario illustrated in Figure 5-5 the *queryHdd* operation is invoked. Each query operation performs a specific query on the supplier product catalogue and stores the matching estimates in an array using the *createEstimates* operation. The product catalogue is queried to return an array of estimates that match the product criteria and order constraints. Finally, the mobile agent filters the best buy from the list of estimates offered by the local supplier, *getSupplierBestBuy*. The *updateBestBuy* operation is invoked to determine if the estimate for the supplier's best buy is competitive. If so, the best buy attribute is updated to reference the supplier estimate.

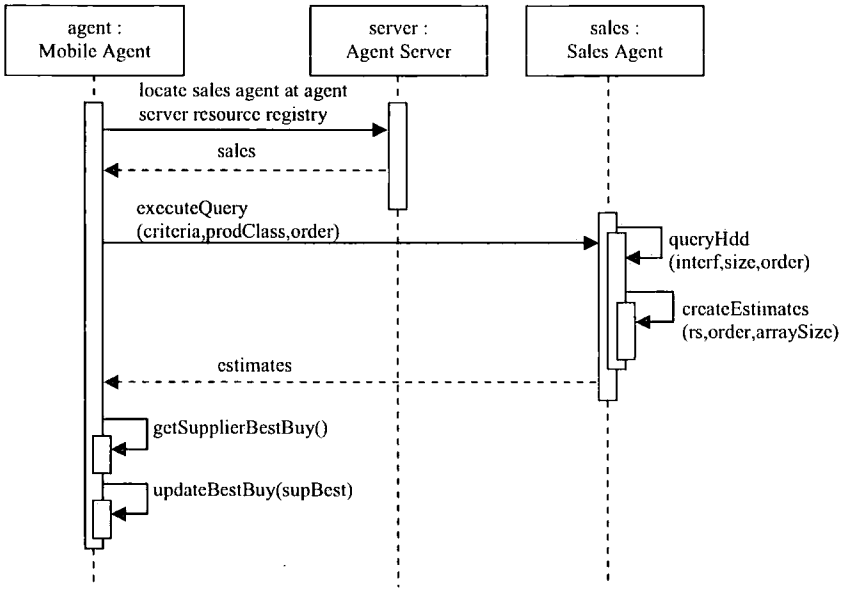


Figure 5-5 UML sequence diagram for sales agent to query product catalogue

2.2 The application interface

Figure 5-6 illustrates the *Driver* and *RunServer* classes of the application interface. Both classes boot an instance of an agent server, represented by the *Server* class that is specific to the mobile agent system adopted in the case study architecture. When the agent server is booted it creates a sales agent resource and registers it with the local registry of software resources.

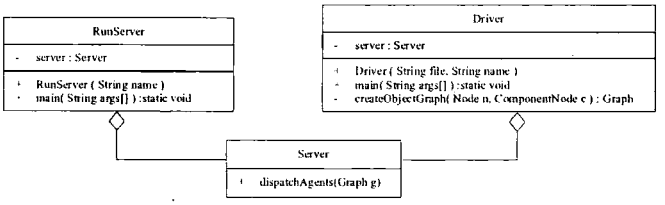


Figure 5-6 UML class diagram for case study application interface

The *RunServer* class boots a basic agent server and creates a sales agent resource. The *Driver* class parses an order XML file and spawns a mobile agent for each product component. Figure 5-7 illustrates a UML sequence diagram for the *Driver* class. The *Driver* class accepts two parameters:

- **XML file**: Path to XML file of product components for stock replenishment.
- **Server id**: Name that uniquely identifies the agent server at the supplier host.

The *Driver* class creates a document object model (DOM) for the order.xml file using the Java Xerces [Xerces04] XML parser. The root node of the DOM is forwarded to the *createObjectGraph* operation to recursively process each node of the order.xml file and create an object graph of product components. Each product component in the object graph is described by: the product class, search criteria, suppliers and order constraints.

An agent server is then booted and assigned the name *ACME*. Most mobile agent systems allow a name to be assigned to the agent server that is incorporated into the full address. For

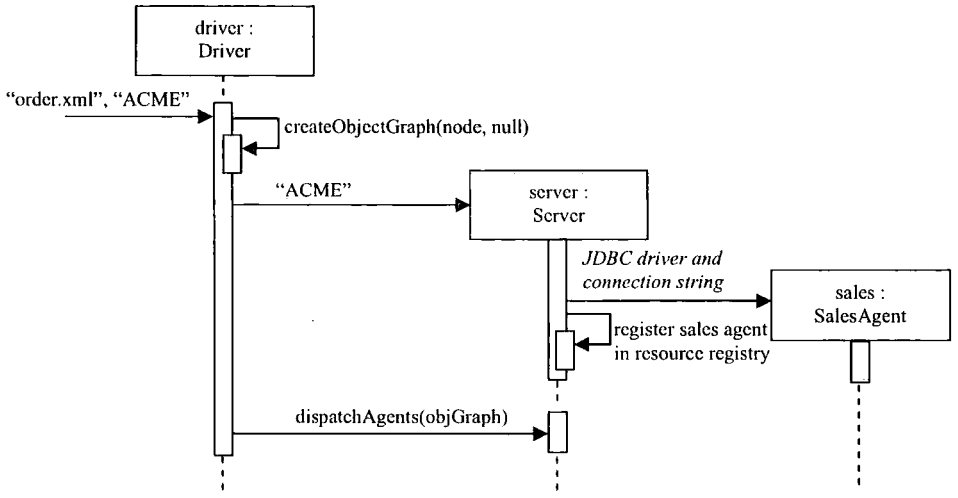


Figure 5-7 UML sequence diagram for driver utility

example, the address of the ACME agent server in IBM Aglets [Oshima98] is atp://cs-128.dur.ac.uk:4044/ACME. The agent server creates and initialises a sales agent instance with the name of the JDBC driver and a connection string for the product catalogue database. The sales agent is then registered with the local resource registry.

Finally, the *Driver* class forwards the object graph of product components to the *dispatchAgents* operation of the agent server. For each product component in the object graph, the *dispatchAgents* operation creates a mobile agent that is initialised with the following parameters:

- Product class.
- Hashtable of product criteria.
- Order constraints.
- Itinerary of suppliers.

Each mobile agent is then dispatched to its itinerary of suppliers.

3 Ajanta case study architecture

3.1 Agent server implementation classes

The sales agent is registered as a *resource* at the Ajanta [Tripathi02] agent server. A resource is an object that acts as an interface to some service or information available at the host [Tripathi02]. In the application case study the sales agent resource provides an interface to the supplier product catalogue database. Resources are stored in a registry at the Ajanta agent server, indexed by a unique URN that is represented in the format *URN:ans:host name/resource name* where:

- Host name is the name of the host where the agent server is running.
- Resource name is a unique name for the resource, e.g. *SalesAgent*.

The Ajanta mobile agent system [Tripathi02] requires that visiting mobile agents know the identity of the resource. Each Ajanta mobile agent can determine the host name property by querying the current agent server environment.

Figure 5-8 illustrates a UML class diagram for the sales agent resource in the Ajanta [Tripathi02] case study implementation. Unshaded classes are implemented by the application developer. The following application classes provide the sales agent resource:

- **SalesAgent**: Interface that defines the operations for the sales agent resource to query the product catalogue. Application developers extend the empty *Resource* interface provided by the Ajanta [Tripathi02] mobile agent system.

- **SalesAgentImpl**: Implementation of the sales agent resource. An Ajanta application resource must inherit from the abstract class *ResourceImpl* and implement the *AccessProtocol* interface. The *AccessProtocol* interface is provided by the Ajanta [Tripathi02] mobile agent system and has a single operation *getProxy* that developers implement to create a proxy to the application resource.
- **SalesAgentProxy**: Contains a transient reference to an instance of the sales agent resource, *SalesAgentImpl*, and implements the *SalesAgent* interface. A proxy forwards requests to the sales agent resource.

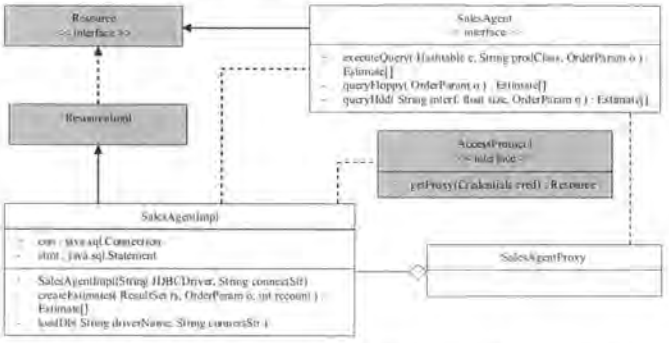


Figure 5-8 UML class diagram for Ajanta sales agent resource

Figure 5-9 illustrates the interactions to register and retrieve a sales agent resource in the Ajanta [Tripathi02] mobile agent system. When the Ajanta agent server is booted, an instance of a sales agent (class *SalesAgentImpl* Figure 5-8) is stored in the resource registry and indexed by a URN identifier (1). Recall that resources in Ajanta [Tripathi02] are retrieved using a URN string that describes the host name and a unique identifier for the resource. Visiting mobile agents send a request to the agent server environment for a proxy to the sales agent resource (2). Included with the request is the URN of the sales agent resource. The Ajanta agent server environment retrieves a reference to the sales agent resource from its resource registry and invokes the *getProxy* implementation of the *AccessProtocol* interface (3). Subsequently, a proxy to the sales agent resource is created (class *SalesAgentProxy* Figure 5-8) and returned to the visiting mobile agent (4). The visiting mobile agent then invokes the *executeQuery* operation on the sales agent proxy to query the product catalogue (5). Consequently, a list of estimates is retrieved from the product catalogue (6).

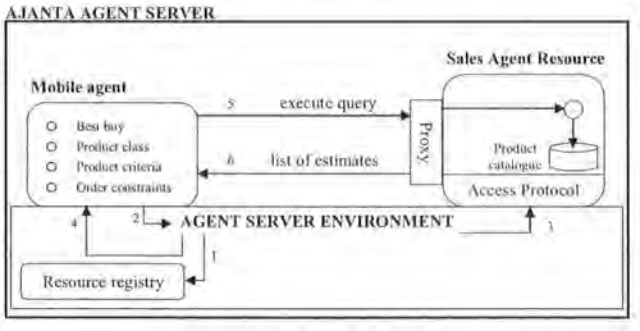


Figure 5-9 Ajanta agent server architecture for sales agent resource

3.2 Ajanta case study mobile agent

Figure 5-10 illustrates the implementation of an Ajanta mobile agent in the case study application. Shaded classes represent those provided by the Ajanta [Tripathi02] mobile agent system. The *ItinAgent* abstract class represents a mobile agent that visits a sequence of agent servers. Subclasses of *ItinAgent* must be initialised with an itinerary and credentials. The *Itinerary* object encapsulates the addresses of agent servers to visit and provides operations to move to the next and previous agent servers in the itinerary. A *Credentials* object migrates with each Ajanta mobile agent. An Ajanta agent server inspects the credentials of a mobile agent to authenticate access to local software resources. Credentials include:

- **Name**: Urn representing the name of the mobile agent.
- **Owner**: Urn of the human user represented by the mobile agent.
- **Creator**: Urn of the application that created and dispatched the mobile agent.
- **Code base**: Urn of the code server for the mobile agent. This defaults to the agent server that created and dispatched the mobile agent.

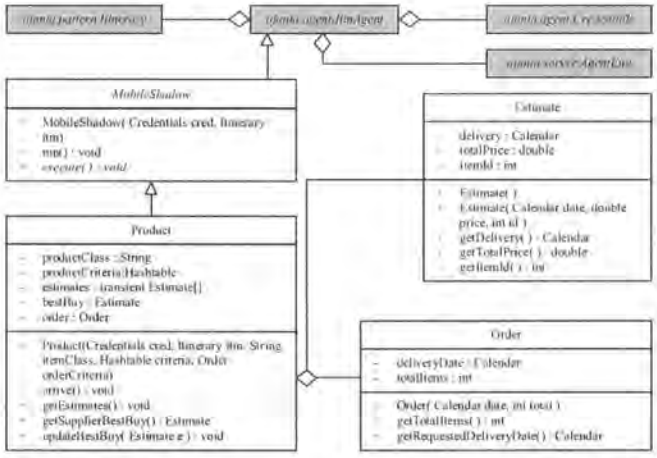


Figure 5-10 UML class diagram for Ajanta case study mobile agent

The *MobileShadow* class represents an Ajanta mobile agent that encapsulates the mobile shadow exception handler. The abstract *execute* operation represents the application specific task performed at each agent server visited in the itinerary. Consequently, subclasses of the *MobileShadow* class inherit the mobile shadow exception handler and override the *execute* operation to implement an application specific task at each agent server. The *Product* class represents the application task for the case study application and has the following responsibilities:

- Interact with the sales agent resource at each agent server environment to retrieve estimates from the supplier product catalogue.
- Log an estimate for the supplier that offers the best buy in terms of delivery date and price.

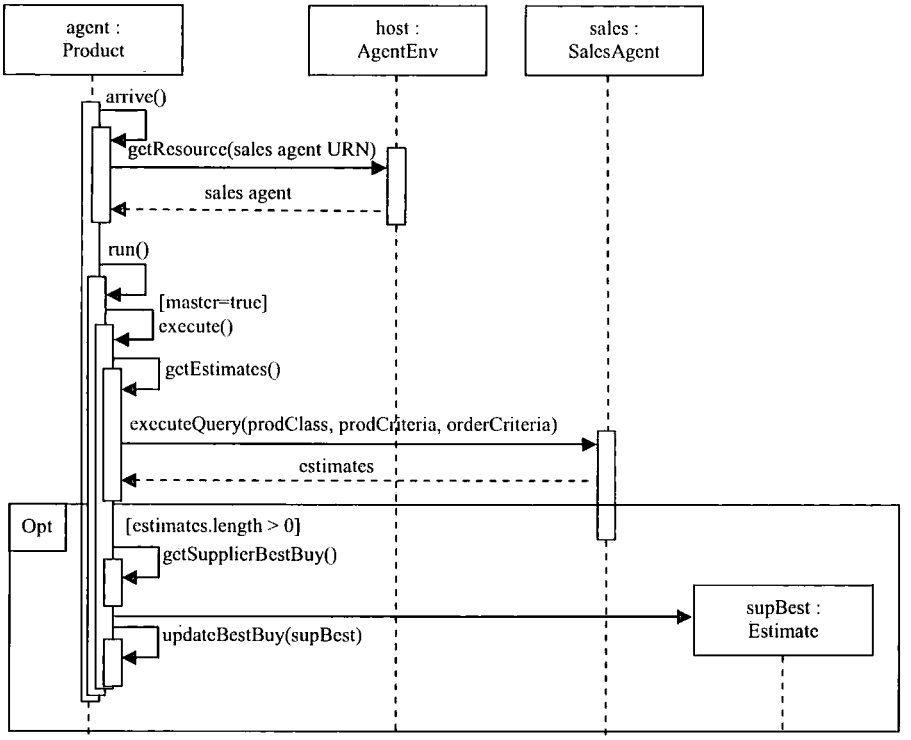


Figure 5-11 UML sequence diagram for Ajanta mobile agent interaction with sales agent

The UML sequence diagram in Figure 5-11 illustrates the interaction between an Ajanta mobile agent (class *Product*) and the sales agent resource at a supplier agent server. When the *Product* mobile agent arrives at the next supplier agent server in the itinerary, it retrieves the local sales agent resource. This is achieved by invoking the *getResource* operation of the Ajanta host environment with the URN of the sales agent resource. All agent servers are assumed to adopt the same name for the sales agent resource.

Subsequently, the main thread inherited from the *MobileShadow* class is executed. Provided that the *Product* mobile agent is a master in the mobile shadow exception handler scheme, the case study application task is started, *execute*. The *getEstimates* operation of the *Product* class retrieves a list of estimates from the supplier's product catalogue by invoking the *executeQuery* operation of the sales agent. The sales agent responds with a list of estimates for matching products in the supplier product catalogue. If matching products were found, i.e. *estimates.length* > *0*, the *getSupplierBestBuy* operation is invoked to determine the best buy offered by the supplier in relation to earliest delivery date and price. The *updateBestBuy* operation is then invoked to determine if the supplier's best buy is competitive. If so, the *Product* mobile agent updates the *bestBuy* field with the supplier's estimate.

4 IBM Aglets case study architecture

4.1 Agent server implementation classes

Figure 5-12 illustrates a UML class diagram for the case study classes that are distributed at an agent server in the IBM Aglets [Oshima98] case study implementation. A *DatabaseManager* agent is created when an agent server is initialised and is responsible for creating a proxy to a sales agent (see class *SalesAgent* Figure 5-12) for each visiting mobile agent. Recall that the sales agent provides an interface to the product catalogue database for visiting mobile agents. A visiting mobile agent uses a sales agent to retrieve a list of matching estimates by sending an *execute query* message with the following parameters:

- **Product class**: A string that represents the class of product, e.g. hard drive.
- **Product criteria**: A hashtable that contains the search criteria for the product, e.g. hard drives may be queried based upon interface type and capacity.
- **Order criteria**: An order object that describes required stock and delivery date.

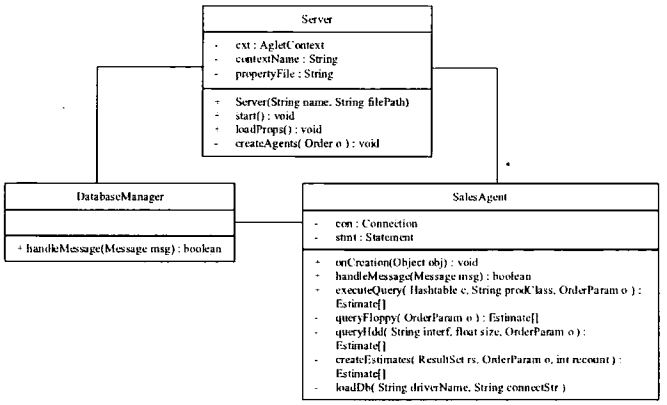


Figure 5-12 UML class diagram for IBM Aglets agent server case study classes

Figure 5-13 illustrates a UML sequence diagram for a sales agent that queries the product catalogue for an IDE hard drive. Upon receipt of an *execute query* message, the sales agent retrieves the product class, product criteria and order criteria arguments from the aglets message object. In response to an *execute query* message, the sales agent invokes the *executeQuery* operation that examines the product class argument to determine which query to perform on the supplier catalogue database. Subsequently, the product criteria are retrieved from the criteria hashtable and forwarded to the appropriate query operation. The *queryHdd* operation is invoked in the scenario illustrated in Figure 5-13.

A sales agent provides operations to query the product catalogue for a given business domain. Each operation performs a specific query on the supplier product catalogue and accepts an *Order* object. For example, the *SalesAgent* class in Figure 5-12 and Figure 5-13 provides the *queryHdd* operation to query the product catalogue for a hard disk drive. The drive interface and

capacity are accepted as parameters. The array of estimates is then sent as a reply to the *execute query* message.

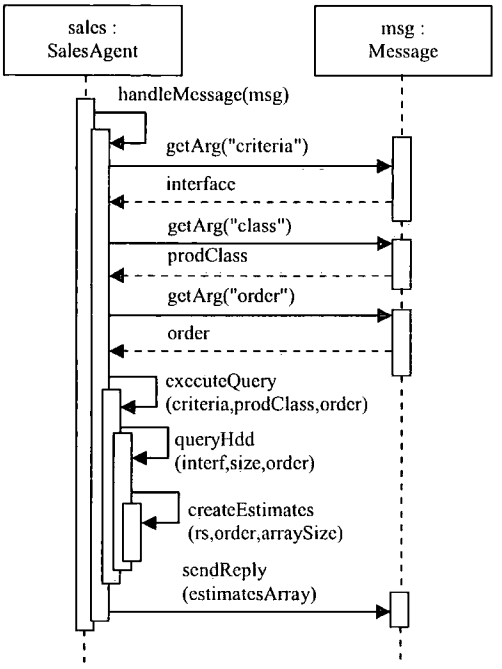


Figure 5-13 UML sequence diagram for Aglets sales agent querying product catalogue

4.2 Aglets case study mobile agent

Unlike Ajanta [Tripathi02], IBM Aglets [Oshima98] does not provide an explicit registry for mobile agents to access software resources at the agent server. However, the agent server environment for IBM Aglets [Oshima98] does allow visiting mobile agents to retrieve a list of executing mobile agents at an agent server. Section 5.2.7 of chapter 4 described the implementation of a simple resource registry that allows IBM Aglets [Oshima98] mobile agents to access, by class name, a stationary mobile agent that acts as a proxy to a specific resource at

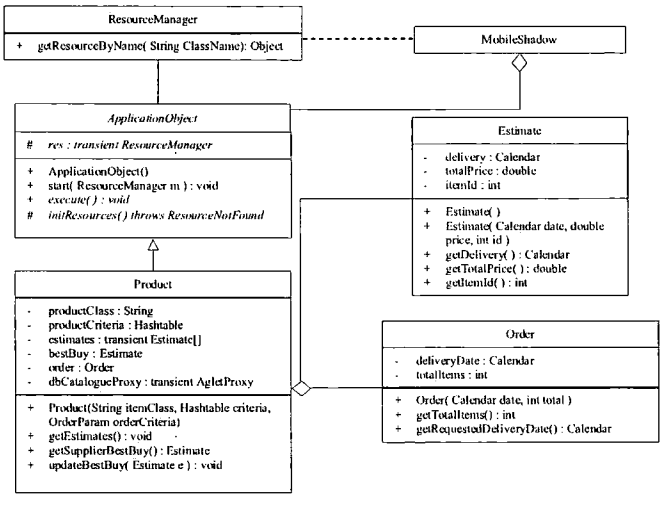


Figure 5-14 UML class diagram for IBM Aglets case study mobile agent

the agent server. Figure 5-14 illustrates the case study application classes that migrate with an IBM Aglets [Oshima98] mobile agent to find the best buy offered by a supplier. A mobile agent (class *MobileShadow*) encapsulates the mobile shadow exception handling scheme and contains an application task (class *Product* derived from *ApplicationObject*), executed at each agent server in the itinerary. The *MobileShadow* class implements the *ResourceManager* interface to allow developers to locate resources, by class name, at the current local agent server.

Figure 5-15 summarises the interactions for retrieving a proxy to a sales agent resource in the IBM Aglets [Oshima98] implementation of the case study. An IBM Aglets [Oshima98] agent server allows visiting mobile agents to retrieve a proxy to any local executing mobile agent. When a mobile agent arrives at a supplier agent server, it examines the list of executing mobile agents, provided by the IBM Aglets [Oshima98] agent server, to retrieve a proxy to the database manager agent. A visiting mobile agent then sends a *connect* message to the database manager to create an instance of a sales agent. The database manager responds by returning a proxy to the new sales agent. Subsequently, the visiting mobile agent sends an *execute query* message to the new sales agent to query the product catalogue.

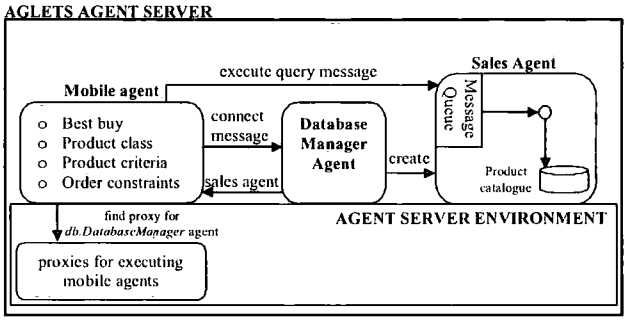


Figure 5-15 IBM Aglets agent server architecture for sales agent resource

The UML sequence diagram in Figure 5-16 illustrates the execution sequence for a visiting mobile agent at a supplier agent server. When the master arrives at a supplier agent server it invokes the *execute* operation. This initializes the application task by invoking the *start* operation of its application object, i.e. the *Product* class. The start operation locates the resources required for execution at the agent server, *initResources*, and triggers execution of the application task, *execute*. In this case study a proxy to the supplier's database manager, *dbProxy*, is retrieved by the *initResources* operation. The database manager is a stationary mobile agent, located at each supplier agent server, that provides visiting mobile agents with a proxy to a sales agent. Subsequently, the *execute* operation is invoked to perform the task of retrieving product estimates and determining the best buy.

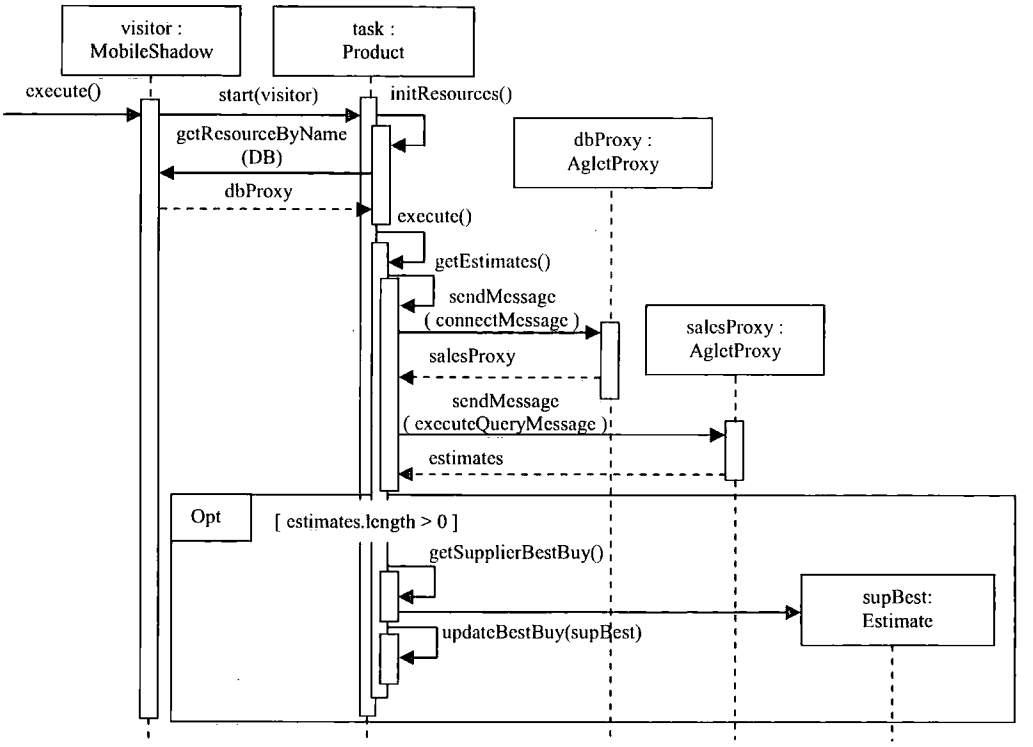


Figure 5-16 UML sequence diagram for Aglets mobile agent interaction with sales agent

The *getEstimates* operation of the *Product* class sends the database manager a connect message to retrieve a proxy to the supplier's sales agent. Recall that the sales agent is a stationary agent located at each agent server that queries the product catalogue. An execute query message is sent to the sales agent, which responds with a list of estimates for matching products in the supplier's product catalogue. If matching products were found, i.e. *estimates.length* > *0*, the *getSupplierBestBuy* operation is invoked to determine the best buy offered by the supplier in relation to earliest delivery date and price. The *updateBestBuy* operation is then invoked to determine if the supplier's best buy is competitive. If so, the *Product* class updates its *bestBuy* field with the supplier's estimate.

5 Summary

This chapter has outlined the design and implementation of an experimental case study environment for the simulation of agent server crash failures and the subsequent analysis of the mobile shadow exception handling scheme. The application domain selected for the case study is information retrieval using an electronic commerce supply chain scenario. A mobile agent is dispatched, for each product component, to several known suppliers to dynamically determine the best buy with respect to delivery date and price.

Chapter 6 Evaluation

1 Introduction

This chapter presents an evaluation of the mobile shadow exception handling scheme detailed in chapter 4. An implementation of the mobile shadow exception handling scheme is first evaluated for the Ajanta [Tripathi02] and IBM Aglets [Oshima98] mobile agent systems. This is performed using the case study application described in chapter 5. The mobile shadow exception handling scheme is then evaluated with respect to the exception handling model for mobile agents detailed in section 2 of chapter 4. Subsequently, the differences between the mobile shadow exception handling scheme and existing systems, for mobile agents to tolerate agent server crash failures, will then be shown. Of particular interest is the suitability of the mobile shadow exception handling scheme for groups of collaborating mobile agents and potential application domains.

2 Application case study evaluation

This section outlines two experiments that were performed using the case study architecture presented in chapter 5. The initial experiment was implemented using the Ajanta [Tripathi02] mobile agent system to compare the performance of the mobile shadow exception handling scheme against a system based upon a timeout mechanism, in the presence of a single faulty agent server crash. Subsequently, an IBM Aglets [Oshima98] implementation was used to investigate the overheads of the mobile shadow exception handling scheme in the event of a random agent server crash.

In both experiments the ping mechanism is implemented using Java TCP sockets. This is due to no support for ICMP packets in Java 1.3. By default Linux kernel 2.4 defines a default of three retry attempts before the TCP protocol reports a failure to the network layer. The timeout value for retransmission (RTO) employed by the TCP protocol is dynamic and is recommended to be initialised to 3s in RFC 1122 (http://www.faqs.org/rfcs/rfc1122.html).

2.1 The Ajanta case study experiment

The aim of the experiment is to use the case study environment to investigate the performance of an Ajanta [Tripathi02] implementation of the mobile shadow exception handling scheme compared to an exception handler that uses a timeout mechanism. In particular the experiment investigated the mobile agent trip time of both schemes for:

- A single agent server crash.
- Normal trip, i.e. no agent server crash failures encountered.

A single mobile agent visited three suppliers to determine the best buy for fifteen 8GB IDE hard drives. For simplicity, each supplier represents the product catalogue using a Mysql 3.23 database with JDBC driver 2.0.8. The experiment was performed on a 10mbps local area network using four 64MB Intel Pentium II 400Mhz (Celeron) PCs running RedHat Linux 7.2 and Ajanta [Tripathi02].

Section 2.1.1 describes the timeout exception handler design. Section 2.1.2 then outlines the performance measurements investigated for both the mobile shadow and timeout exception handler schemes. Finally, section 2.1.3 presents the experiment results and analysis.

2.1.1 The timeout exception handler

The timeout exception handler is located at the home agent server and is associated with a group of mobile agents dispatched to perform an information retrieval task. The timeout exception handler (Figure 6-1) waits for a timeout period and then resends mobile agents that did not return.

```
while !dispatch.isEmpty() {//while agents to send}
  handlerTimeOut(t) {// wait t seconds}
  handlerResend(dispatch) {//send replacements}
```

Figure 6-1 Timeout scheme pseudocode

The following meta operations and state are provided at the home agent server:

- *Dispatch:* List of dispatched mobile agents.
- *Task():* Create dispatch list and send a group of mobile agents to perform an information retrieval task.
- *Add(A, dispatch):* Add a mobile agent *A* to the dispatch list.
- *Remove(A, dispatch):* Remove mobile agent *A* from the dispatch list. A mobile agent is removed from the dispatch list when it returns home.
- *Handler():* Crash exception handler that executes after task operation completed.
- *HandlerTimeOut(t):* Wait for *t* seconds.
- *HandlerResend(dispatch):* Resend mobile agents in dispatch list.

The timeout exception handler tolerates any number of agent server crash failures with no additional overheads imposed on mobile agents and remote agent servers. However, in the event of an agent server crash, all agent servers are revisited. Furthermore, it is difficult to select a timeout value in asynchronous systems since there are no established boundaries for processor speed and communication delay. If an aggressive timeout value is used, many duplicate agents are dispatched, e.g. a mobile agent may not return within the timeout period if it executes at one or more slow agent servers. If a conservative timeout value is chosen, the application will block until the timeout expires. This occurs even when some mobile agents return.

2.1.2 Performance measurements

Two performance measures were obtained for both exception handler schemes:

- **Normal round trip time:** The time taken for a mobile agent to complete its itinerary and return to the home agent server with the best buy. Agent servers visited by the mobile agent suffer no crash failures.
- **Crash round trip time:** The time taken to complete an itinerary and report back to the home agent server with the best buy in the presence of a single agent server crash.

Furthermore, the performance overheads in Figure 6-2 were measured for a single agent server crash. Path 1-2-3-1 for the mobile shadow exception handling scheme represents handler execution for an agent server crash.

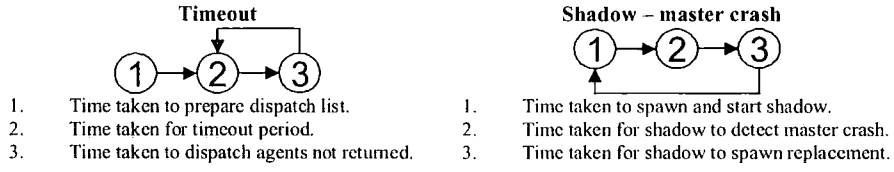


Figure 6-2 Performance overheads for Ajanta case study experiment

The source code to calculate the performance overheads for the timeout exception handler scheme imposed no additional increase to the normal or crash round trip times. This is because the calculations are performed at the agent server that dispatched the mobile agent. However, this is not true for the mobile shadow exception handling scheme, since the mobile agent state must be augmented to log the times for spawning a shadow and detecting the crash of the agent server where the master is located. To provide an accurate comparison there were two sets of round trip times for the mobile shadow exception handling scheme:

1. Round trip times assume augmented mobile agent state with performance variables.
2. Round trip times assume no augmented mobile agent state, i.e. no performance variables.

Conservative timeout values were selected for the timeout scheme so that a mobile agent has sufficient time to return before another replica is dispatched.

2.1.3 Results and analysis

The round trip times were obtained from forty trial runs. A new mobile agent was dispatched for each trial run. To simulate an agent server crash a mobile agent waits at a specific agent server to enable manual termination. After recovery the crashed agent server was restarted. This was done due to the enforced security model of Ajanta [Tripathi01], i.e. there is

no mechanism to enable a mobile agent to terminate an agent server, or for an agent server to halt, when a specific mobile agent arrives.

Figure 6-3 illustrates the time increase introduced by the crash round trip for each exception handler. Performance calculations for the mobile shadow exception handling scheme impose a minor increase of 0.50% on the round trip times. The trip times for the mobile shadow scheme represent the case where the mobile agent state is not augmented with performance calculations.

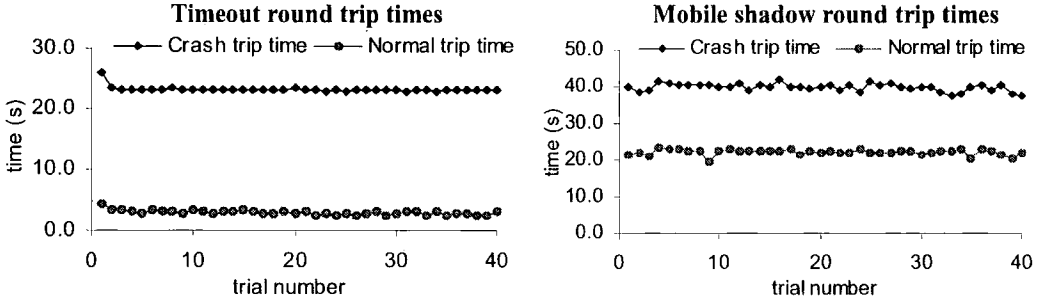


Figure 6-3 Trip times for timeout and mobile shadow exception handler

The timeout exception handling scheme offers a quicker average normal and crash round trip, i.e. 3s and 23.2s respectively. This is compared to the mobile shadow exception handling scheme that provides an average of 22.2s and 39.6s. However, the crash trip time for the timeout scheme depends upon the total agent servers visited. Longer trips need a larger timeout, increasing the crash round trip time. The mobile shadow exception handling scheme is independent of trip length and consequently may perform better for longer trips.

Figure 6-4 illustrates performance overheads for the timeout exception handler. The timeout exception handler has insignificant times for initialising a record of agents to dispatch (0.1ms) and resending those agents (0.6s) that failed to return. Both measures fall within 1 second.

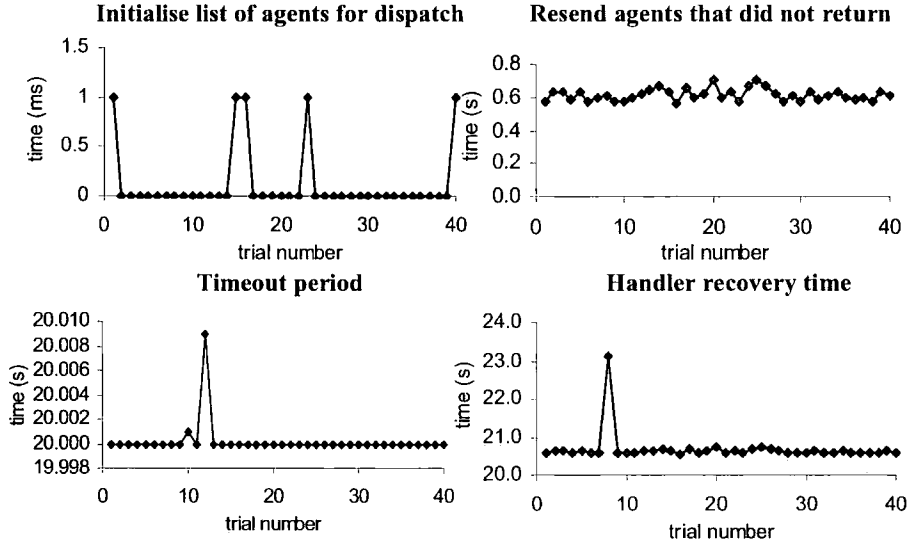


Figure 6-4 Timeout exception handler overheads

However these measures were obtained from dispatching a single mobile agent. The major performance overheads for the timeout exception handler are the timeout period (20s) and recovery time (20.7s). The recovery time represents the time to resend agents that have not returned in addition to the timeout period that elapsed when the handler completed.

The mobile shadow exception handling scheme offers slower round trip times. Figure 6-5 illustrates the performance overheads. A shadow starts a thread to ping the next agent server where its master will execute. The average time for a shadow to be notified of its master's agent server crash is 8.3s. The average time to spawn and start a shadow under normal execution is negligible, i.e. 0.8s. A larger overhead (13.4s) is introduced when a shadow spawns and starts its own shadow during recovery. This may be because the shadow migrates while its child pings its destination agent server.

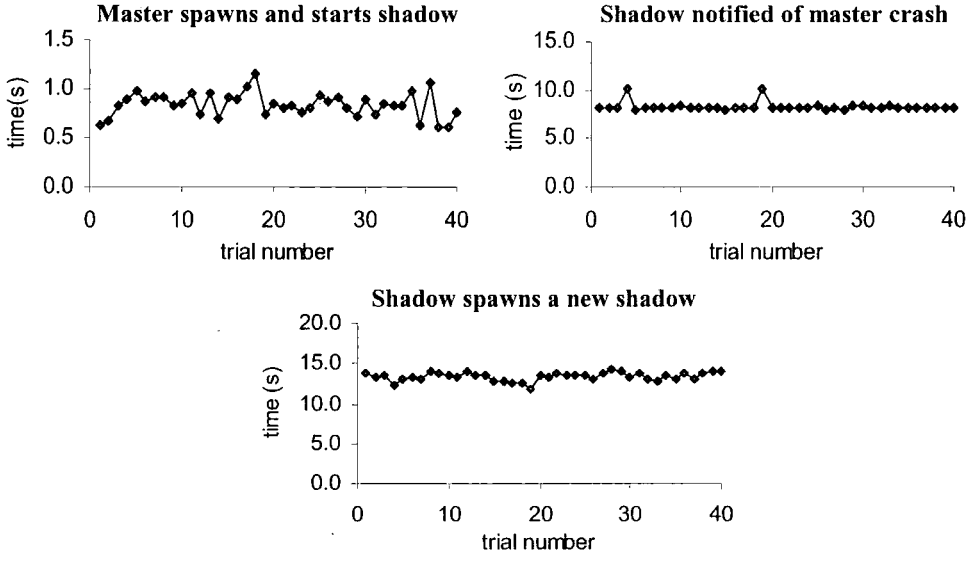


Figure 6-5 Mobile shadow overheads

To summarise, the timeout exception handling scheme offers quicker round trip times. However, it is not independent from the itinerary size, i.e. a larger timeout value is necessary to wait for mobile agents to return from longer trips. The advantage of the mobile shadow exception handling scheme is that it is independent from the itinerary size. This is because the shadow is activated, at the last available preceding server visited, in the event of an agent server crash. Furthermore, the mobile shadow exception handling scheme offers the smallest trip time increase in the event of an agent server crash.

2.2 The IBM Aglets case study experiment

The experiment outlined in section 2.1 investigated the mobile agent trip times for the mobile shadow exception handling scheme and a timeout exception handler. The trip times were obtained for the crash of a specific agent server. The security model of the Ajanta [Tripathi02] mobile agent system does not allow simulation of a random agent server crash, since there is no

mechanism available for the agent server environment to react to the arrival of mobile agents. However, in a wide area network environment an agent server crash is random. Consequently, the experiment uses the IBM Aglets [Oshima98] case study implementation to analyse the performance of the mobile shadow exception handling scheme with a single random agent server crash.

The same experimental environment outlined in section 2.1 was adopted for the case study environment. A single mobile agent visits three suppliers to determine the best buy for fifteen 8GB IDE hard drives. For simplicity, each supplier represents its product catalogue using the Mysql 3.23 database system with JDBC driver 2.0.8. The experiment was performed on a 10mbps LAN using four 64MB Intel Pentium II 400Mhz (Celeron) PC's running RedHat Linux 7.2 and IBM Aglets [Oshima98]. Section 2.2.1 describes the simulation of an agent server crash. Section 2.2.2 then outlines the performance measurements investigated. Finally, section 2.2.3 presents the experiment results and analysis.

2.2.1 Simulating a random crash

To simulate a crash, the mobile agent terminates an agent server using the Java command *System.exit(1)*. Permission to terminate the Java Virtual Machine is assigned in the security policy file for each agent server. The *CrashSimulator* class (Figure 6-6) is initialised with the total agent servers to visit (*iTripSize*). A random number (*iCrashIndex*) represents the n^{th} visited agent server that will crash, i.e. *0 < iCrashIndex <= iTripSize*. Before a mobile agent migrates to the next agent server in its itinerary it increments the total number of agent servers visited (*iVisited*), i.e. *crash.increment()*. When execution begins at the next agent server, the mobile agent determines if it is at the agent server selected to crash (*iVisited==iCrashIndex)*. This is done by invoking *CrashSimulator.crash()*. If so, the mobile agent terminates the agent server.

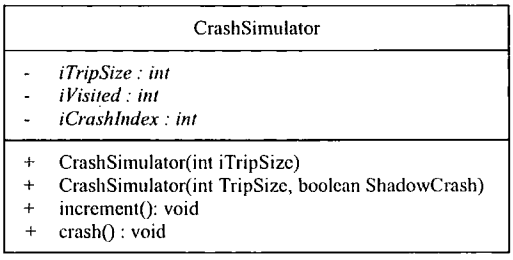


Figure 6-6 CrashSimulator class

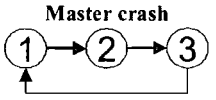
The crash simulation above is applicable to a master crash. To simulate a shadow crash the master delegates the responsibility of terminating the agent server to its shadow. Furthermore, a random number must be generated in the range 0 < iCrashIndex <= iTripSize-1 since it is assumed that a master discards its shadow when it returns to the home agent server. Consequently, the *CrashSimulator* class provides a second constructor *CrashSimulator(TripSize, ShadowCrash)*.

2.2.2 Performance measurements

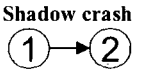
Two performance measures were obtained for an insight into the overheads introduced by the mobile shadow exception handling scheme in the event of a random agent server crash:

- **Normal round trip time**: Time taken to complete an itinerary and return to home agent server with the best buy. Visited agent servers suffer no crash failures.
- **Crash round trip time**: Time taken to complete an itinerary and return to home agent server with the best buy in the presence of a single agent server crash.

Furthermore, the performance measures in Figure 6-7 were obtained in the event of a single agent server crash.



1. Time taken to spawn and start shadow.
2. Time taken for shadow to detect master crash.
3. Time for shadow to spawn replacement.



1. Time for master to detect shadow crash.
2. Time for master to spawn replacement shadow.

Figure 6-7 Performance overheads for IBM Aglets case study experiment

To simulate a random agent server crash the state of the mobile agent was augmented with an instance of the *CrashSimulator* class in addition to code for performance measurements. To provide accurate normal and crash round trip times there were two sets of normal trip times:

1. Normal round trip time assumes an augmented mobile agent state that includes an instance of the *CrashSimulator* class and performance variables.
2. Normal round trip time assumes no augmented mobile agent state.

2.2.3 Results and analysis

The normal and crash round trip times were obtained from forty trial runs. A new mobile agent was dispatched for each trial run. An agent server crash simulation was reset before the next trial.

A comparison of round trip times for the crash of a master and a shadow is illustrated in Figure 6-8. Performance calculations for the mobile shadow scheme imposed a minor increase of 0.50% on the round trip times. The normal round trip time assumes no state augmentation for performance calculations or the *CrashSimulator* class.

The mobile shadow exception handling scheme provides an average normal round trip time of 2.1s. When fault tolerance measures are exercised in the presence of a random agent server crash, the round trip time significantly increases. A shadow crash offers a quicker round trip time of 13.8s (11.7s increase) compared to 14.6s (12.5s increase) for a master crash. This is due

to the way in which the shadow crash is simulated. When the master migrates to the next agent server in the itinerary the shadow terminates its agent server. Consequently, when the master arrives at the next agent server in the itinerary its ping thread detects the crash of the shadow.

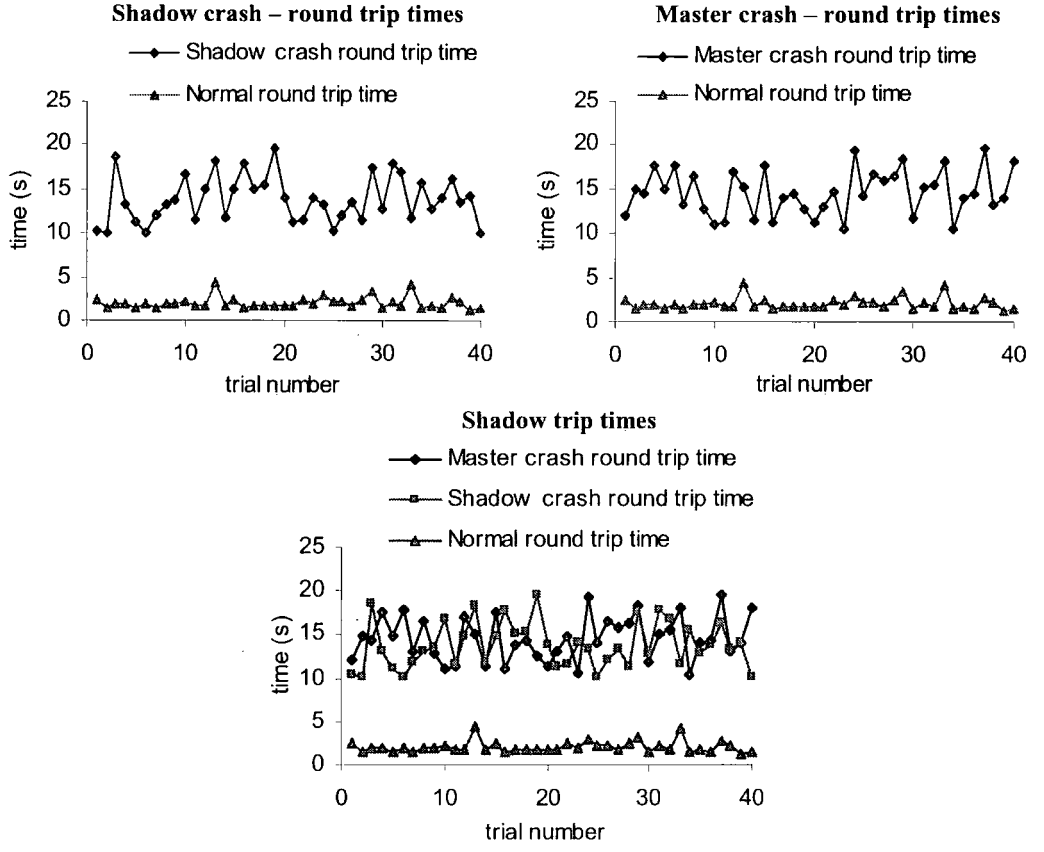


Figure 6-8 Mobile shadow trip times

Figure 6-9 illustrates the performance overheads for master and shadow crashes. The time taken to spawn and start a shadow is negligible for both a master and shadow crash. For example, when handling a master crash the average time for a shadow to spawn a new shadow is 76.5ms (0.08s).

The largest overhead is the time taken for a shadow to be notified of its master's agent server crash (4.4s). This is explained by the concurrent execution of the shadow and its ping thread. Every shadow starts a thread to ping its master's agent server. The ping thread pings until it detects a crash and notifies the blocked shadow. Similarly, the master pings its shadow concurrently.

To summarise, quicker round trip times were obtained with the IBM Aglets [Oshima98] case study design for a random agent server crash. This is attributed to the design of the crash simulation and the use of JDK1.3 with the IBM Aglets [Oshima98] implementation.

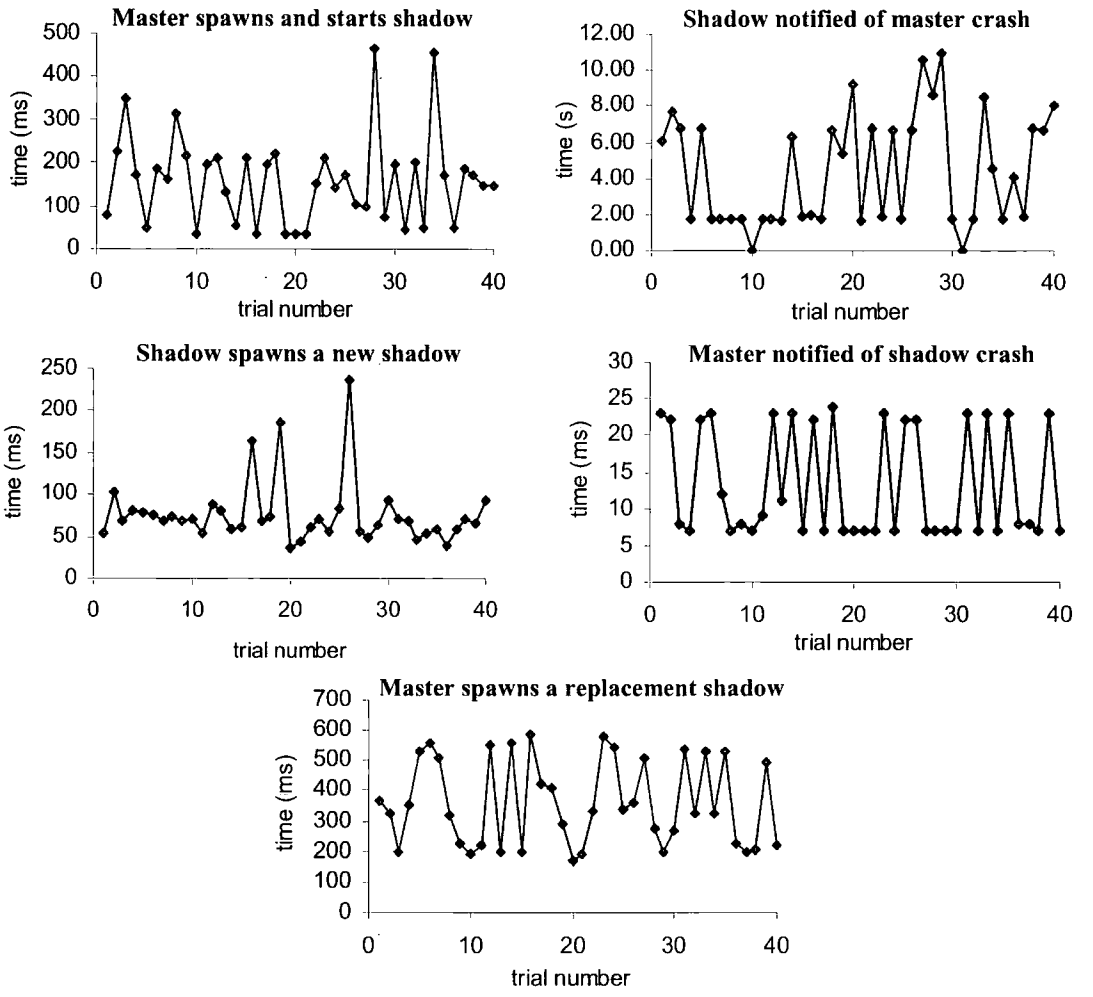


Figure 6-9 Performance overheads for mobile shadow exception handler

Furthermore, the largest overhead was found to be the time taken for a shadow to detect its master crash. This is attributed to the concurrent execution of the shadow and its ping thread.

2.3 Summary

The experiment performed using the Ajanta [Tripathi02] case study environment compared the mobile shadow exception handling scheme with an exception handler that uses a timeout mechanism to detect agent server crash failures. Systems that employ a timeout mechanism must use a timeout period that exceeds the slowest network conditions for the mobile agent trip time. Consequently, even if there are no agent server crash failures, the timeout period must still run to completion to determine failure. The advantage of the mobile shadow exception handling scheme is that no timeout mechanism is necessary for the mobile agent trip length. Consequently, in the event that the agent server occupied by the master crashes, the shadow is available as a replacement.

There are a number of extensions that can be performed for the experiment. Firstly, a greater number of trial runs can be performed. Currently, an agent server must be manually

restarted, for each run, when measuring the crash round trip time. A greater number of trial runs could be obtained if there was the ability to automatically restart an agent server process that encountered a crash failure. Secondly, the experiment could vary the size of the mobile agent and the total number of random agent server crashes encountered during a trip. The size of the mobile agent could be increased by using redundant state or a larger itinerary.

A final consideration would be to deploy the case study architecture in a wide area network environment. Currently, both of the experiments outlined in sections 2.1 and 2.2 are deployed in a local area network. Consequently, the results are applicable to mobile agents that are deployed within the same administrative domain to compress information distributed within the local network. The next step would be to investigate the performance of the mobile shadow exception handling scheme in a wide area network for large scale distributed information retrieval. This would have to be performed in collaboration with other institutions. Alternatively, a simulation of a wide area network environment could be investigated.

3 Evaluation using exception handling model

This section evaluates the mobile shadow exception handling scheme with respect to the conceptual exception handling model outlined in chapter 4, section 2. The mobile shadow exception handling scheme provides a fault tolerant service for maintaining mobile agent availability in the presence of agent server crashes. This service is embedded within the application mobile agent. Recall that the mobile shadow scheme employs two mobile agents:

- **Master**: Responsible for executing the application task.
- **Shadow**: Replica that monitors the agent server currently occupied by the master. The shadow is located at the agent server previously visited by the master.

The mobile agent exception handling model partitions the behaviour of a mobile agent into *normal* and *exceptional*. With respect to the mobile shadow exception handling scheme, the normal behaviour of a mobile agent corresponds to execution of an application task at each agent server visited in the itinerary. The exceptional behaviour is triggered when an application exception is signalled through interaction with software services at the agent server. Alternatively, a mobile agent's exceptional behaviour may be triggered due to environmental conditions. For example, the agent server may disallow mobile agent execution due to inadequate access privileges. Furthermore, an agent server may signal an exception to a mobile agent when it failed to dispatch the mobile agent to the next agent server in the itinerary. For example, the destination agent server may have failed by crashing.

The mobile shadow exception handling scheme specifically handles exceptions raised by the agent server environment with respect to an agent server crash. Consequently, exceptional

behaviour is triggered when an agent server occupied by the master or shadow fails by crashing. The application developer is expected to address application exceptions raised by software services at agent servers visited in the itinerary. Consequently, the developer of a mobile agent must be aware of the exceptions that are signalled by software services at agent server environments visited in the itinerary. This appears to be the normal case for mobile agents in the distributed systems community. To the author's knowledge Ajanta [Tripathi02] is the only mobile agent system that imposes a framework to handle application exceptions for mobile agents.

There are two exception handlers that migrate with a mobile agent in the mobile shadow scheme:

- **Master exception handler**: Replaces a master lost due to an agent server crash.
- **Shadow exception handler**: Replaces a shadow lost due to an agent server crash.

Each agent server provides a ping service, s_{ping}, to respond to ping messages from remote mobile agents. This is synonymous with a service at an agent server in the conceptual exception handling model presented in chapter 4. If there is no ping response from the agent server's ping service a failure exception is raised in the master or shadow. At this point the exception handling model presented in chapter 4 offers the following options for mobile agents that receive a failure exception from a service at a remote agent server:

- Retry.
- Report back to home agent server.
- Report back to parent mobile agent.

If the shadow exception handler is triggered a retry action is performed. The master raises a failure exception when the ping service located at the shadow's agent server fails to respond. Consequently, the master issues a retry action by dispatching a shadow to the next available agent server that was previously visited.

The master exception handler is triggered when the ping service located at the master's agent server fails to respond. In this scenario the shadow reports back to the home agent server when the trip is complete. The IBM Aglets [Oshima98] implementation for the mobile shadow exception handling scheme, presented in chapter 4, skips the agent server that crashed. Consequently, a replacement master arrives at the next available agent server in the itinerary. This policy is applicable for applications that require the information within a time deadline. For example, resending a mobile agent to visit failed agent servers may add significant overheads to the time requirements imposed by the application. A retry action for the master exception handler was considered whereby the replacement master installs a clone that repeatedly retries

visiting the crashed agent server. Obviously this scheme requires a shadow to monitor the availability of the agent server occupied by the clone. However, the disadvantages are as follows:

- The state of the mobile agent is increased to log the id and location of clones that attempt to visit a crashed agent server.
- The scheme requires two additional mobile agents. A clone is required for dispatch to the crashed agent server. Furthermore, a shadow must monitor the availability of the clone. Eventually, the performance of agent servers could be degraded, especially if recursion occurs due to a clone failure.

In the worst case scenario a mobile agent may visit none of the hosts in its itinerary if all have crashed. One solution is to use an itinerary pattern [Tripathi02] whereby the mobile agent logs the success for each itinerary entry. The home agent server can take appropriate action when the mobile agent returns, e.g. a mobile agent may be dispatched to all failed agent servers. The guardian exception handling model presented in [Tripathi02] provides a *guardian* at the home agent server that encapsulates recovery behaviour for exceptions that cannot be handled by mobile agents. The guardian may also be used to co-ordinate recovery of mobile agent groups. In the exception handling model for mobile agents (presented in chapter 4, section 2), the guardian may encapsulate the recovery behaviour at the home agent server when a mobile fails to retry or locate an alternative software service.

The mobile shadow exception handling scheme is evidently recursive for the crash of a master or shadow. For example, in the event of a master crash the shadow spawns a new shadow and then replaces the master. The case study application presented in chapter 5 and the IBM Aglets [Oshima98] implementation in chapter 4 assumes that a single mobile agent visits a sequence of agent servers. No children are spawned to perform a task on the behalf of the mobile agent. This scenario is raised in the exception handling model presented in chapter 4, section 2. The mobile shadow exception handling scheme addresses this scenario provided that there is no synchronous relationship introduced between a parent and its children. For example, a child may either report back to the home agent server or perform a task on the behalf of its parent and then terminate. However, if a parent requires a response from a child then a synchronous relationship is introduced and the mobile shadow exception handling scheme must be extended. If the agent server occupied by a master crashes, then children are not aware of the location of the replacement master. Similarly, a replacement master is not notified when a child has completed its task. A possible solution would be to impose the restriction that all children report back to the home agent server. However, fault tolerance mechanisms are required to provide high availability of the home agent server.

To summarise, the mobile shadow exception handling scheme provides the foundation for using the exception handling model in information retrieval environments where mobile agent loss must be tolerated. The scheme offers two advantages. Firstly, mobility and replication provide fault tolerance during the mobile agent trip, i.e. an exception handler that is independent of trip length migrates with the mobile agent. This is compared to a timeout scheme [Pears03] where the application developer must vary the timeout interval for different trip lengths. Secondly, the scheme is useful for groups of collaborative information retrieval mobile agents, since the master and shadow comprise a single fault tolerant group member. Alternative schemes exist, e.g. [DeAssisSilva00, Pleisch03, Schneider97, Strasser99], that replicate a mobile agent at each stage of its itinerary to an anticipated set of agent servers. However, this induces a complex design for collaborative information retrieval mobile agents. For example, a single group member requires that replicas are deployed, at alternative agent servers, for each stage of the itinerary.

4 Feature-based evaluation

In addition to the evaluations using the case study and exception handling model, the mobile shadow exception handling scheme was compared with other existing systems for mobile agents to survive agent server crashes. This was done to highlight the similarities and differences against existing research.

The features identified, and the rationale for selection, are described in the following table.

Feature	Explanation
Fault tolerance location	This defines where the fault tolerance design is located. If located within the mobile agent then there is the danger that the state of the mobile agent exceeds bandwidth limitations. If located within the agent server then it must be feasible to adopt the algorithm for all mobile agent systems.
Recovery mechanism	This defines the recovery mechanism. A forward recovery mechanism implies that modifications to the state of the agent server are not undone. The application developer is responsible for compensating the state of the agent server if it recovers from the crash. If backward error recovery is employed it is possible to undo actions performed by mobile agents.
Stable storage	This defines if mobile agents are stored in stable storage at each agent server. This is important since it describes if mobile agents are restarted by the agent server when it recovers from a crash.

Structure to determine alternative agent server	This defines if the fault tolerance design provides any structure for mobile agents to visit alternative agent servers in the event of an agent server crash. Alternatively, this responsibility may be left with the application developer.
Communication assumptions	This defines the assumptions made by the fault tolerance design regarding the underlying network conditions. Consequently, this is important to potential users since it describes the underlying conditions of the mobile agent's network environment for correct execution of the fault tolerance protocol. If no communication assumptions are made, then mobile agents can survive network partitions and transport failures due to unreliable communication links.
Fault tolerance mechanism	This defines the fault tolerance mechanism used for mobile agents to survive agent server crash failures. A spatial replication mechanism [DeAssisSilva00, Pleisch03, Schneider97, Strasser99] dispatches replica mobile agents to a group of agent servers for the next stage in the itinerary. A temporal replication approach executes a mobile agent at a single agent server. If execution fails the mobile agent is dispatched to another agent server.
Child failure	This defines if children spawned, by a parent mobile agent, can survive an agent server crash. "Asynchronous" indicates that children survive agent server crashes, provided there is no need to report back to the parent. "Synchronous" indicates that children survive mobile agent crashes and report back results to the parent.

Table 6-1 Features to be identified in the feature analysis

Table 6-2 shows the features contained within existing systems for mobile agents to survive agent server crash failures. The mobile shadow exception handling scheme has also been included for reference.

System	Fault tolerance location	Recovery mechanism	Structure to determine alternative agent server	Stable storage	Communication assumptions	Fault tolerance mechanism	Child failure
Mobile shadow [Pears03]	Mobile agent	Forward	Developer defined	No	Reliable messaging assumed	Temporal replication	Asynchronous
FATOMAS [Pleisch01]	Mobile agent or agent server	Forward	Developer defined	Yes	None	Spatial replication	Synchronous and asynchronous
GMD FOKUS [DeAssisSilva00]	Agent server	Forward and backward	Developer defined	Yes	Reliable messaging assumed	Spatial replication	Synchronous and asynchronous
JAMES [Silva00]	Hybrid	Backward	Yes	Yes	None	Temporal replication	Asynchronous
Mole [Strasser98]	Agent server	Forward	Developer defined	Yes	Reliable messaging assumed	Spatial replication	Asynchronous
NAP [Johansen99]	Agent server	Backward	Developer defined	Yes	Reliable messaging assumed	Temporal replication	Asynchronous
Net Pebbles [Mohindra00]	Agent server	Backward	Yes, language constructs for alternative agent servers	Yes	None	Temporal replication	Asynchronous

Table 6-2 Feature analysis of fault tolerance systems for surviving agent server crashes

The ideal fault tolerant scheme for mobile agents to survive crash failures depends upon the requirements of the application and the scale of the network. Consequently, emphasis must be given to specific tasks, with the recognition that this may require compromise dependent upon the application domain.

Information retrieval mobile agents perform idempotent operations at each agent server visited in the itinerary, i.e. a mobile agent that interacts with software services at each agent server does not modify the application state of the agent server. Mobile agents only consume information from agent servers visited in the itinerary, thus eradicating the need for backward recovery mechanisms. Furthermore, if the mobile agent is dispatched in a wide area network to retrieve information then the protocol ideally migrates with the mobile agent. Interoperability between mobile agent systems is still a significant problem. However, there is evidence of research interest to tackle the problem [Brazier02, Grimstrup02, Misikangas00, Pinsdorf02]. If the fault tolerance protocol migrates with the mobile agent then fault tolerance is provided irrespective of the installed mobile agent system. This increases the interoperability of the protocol with mobile agent systems, i.e. enterprises do not have to modify the agent server environment. However, the increase of the mobile agent's state must be kept to a minimum.

Mobile agents may also be used to perform an application specific task on the behalf of a user. In this scenario the mobile agent may modify the state of agent servers. For example, a mobile agent that purchases a product at an agent server alters the stock level. Consequently, backward recovery mechanisms are necessary to rollback the application state of the agent server in the event that a crashed agent server eventually recovers. Alternatively, forward recovery mechanisms [Pears03, Pleisch01, Strasser98] may be used to compensate the actions performed by the mobile agent, e.g. cancelling the purchase made by the mobile agent.

If the application is deployed over the wide area network then a system is preferred that does not make any assumptions regarding communication in the network. For example, reliable messaging is a weak assumption for applications that are deployed in large scale networks where network partitions are frequent. In this scenario the mobile shadow exception handling scheme fails if the shadow and master are separated by network partitions. Consequently, the shadow believes that the master has failed due to an agent server crash. This becomes a problem with applications that require exactly once execution semantics since there are two instances of the application mobile agent, separated by network partitions. However, this is not a problem if the mobile shadow exception handling scheme is used for information retrieval applications where the duplicate mobile agent only consumes information.

Finally, if the mobile agent application deploys a mobile agent that dispatches children to perform an application task then the mobile shadow exception handling scheme is suitable,

provided that there is no synchronous relationship introduced between the parent and child. If it is necessary for the parent to synchronise with its children, e.g. to retrieve results, then the systems described in [DeAssisSilva00, Pleisch01] are preferred.

To summarise, the mobile shadow exception handling scheme is ideally suited to information retrieval applications since no backward recovery is necessary. The strength of the mobile shadow exception handling scheme is that a mobile agent is a single fault tolerant entity that can survive agent server crash failures. The fault tolerance protocol migrates with the mobile agent, eradicating the need to modify the agent server environment at remote hosts. Other systems that employ temporal monitoring require modification of the agent server environment for installation of fault tolerance measures. Consequently, the mobile shadow exception handling scheme has the potential to ease the complexity introduced for a group of mobile agents deployed for information retrieval. This is compared to systems that employ spatial replication to tolerate agent server crashes. Systems that use spatial replication require a group of agent replicas for each mobile agent in the group. The group of replicas are dispatched to a set of discrete agent servers for each stage in the itinerary. Additionally, a voting algorithm is necessary for replicas to agree upon the following points:

- A mobile agent has failed due to an agent server crash.
- The identity of the mobile agent that is executing the application task.

5 Summary

This chapter has provided an evaluation of the mobile shadow exception handling scheme detailed in chapter 4. The implementation of the mobile shadow exception handling scheme was evaluated using a number of techniques.

Firstly, the implementation of the mobile shadow exception handling scheme was evaluated for the Ajanta [Tripathi02] and IBM Aglets [Oshima98] mobile agent systems. The mobile shadow scheme has the advantage that it is independent of the itinerary trip length. Consequently, a performance benefit is offered over systems that use a timeout mechanism at the home agent server, for larger itinerary lengths. Furthermore, performance overheads were also obtained for an Ajanta and IBM Aglets [Oshima98] implementation of the mobile shadow exception handling scheme.

Secondly, the mobile shadow exception handling scheme was justified with respect to the exception handling model presented in chapter 4, section 2. This highlighted the application exception handling options available with respect to an agent server or host crash. Furthermore, the difficulties encountered for tolerating a child and parent crash failure while preventing blocking is highlighted.

Finally, a comparison was provided with existing systems. The result of the analysis highlights that the mobile shadow scheme is applicable to idempotent applications that operate within a closed network environment. The issues with using the scheme in a wide area network environment were highlighted. Furthermore, it was observed that the mobile shadow exception handling scheme has the potential to ease the complexity introduced for a group of mobile agents deployed for information retrieval, compared with systems that use spatial replication to tolerate agent server and host crash failures. Reducing the additional complexity introduced by fault tolerance is an important issue for mobile agent applications, in order to preserve the potential savings in bandwidth.

Chapter 7 Conclusions

1 Introduction

The aim of this research was to investigate and develop a scheme that uses exception handling for mobile agents to survive agent server crash failures. Therefore, a major part of the thesis is firstly a background of mobile agents and techniques for exception handling in traditional distributed systems. Subsequently, existing systems for maintaining the availability of mobile agents against agent server crashes were investigated. The problem of maintaining the availability of mobile agents against agent server crashes is a difficult one. Solutions use temporal or spatial replication mechanisms. The remainder of the thesis addresses the problem by firstly proposing an exception handling model for mobile agent systems. Consequently, based upon the understanding gained, a scheme was implemented that uses a temporal replication solution. This was then deployed within a case study application.

The mobile shadow exception handling scheme proposed in this thesis uses temporal replication for mobile agents to survive agent server crash failures. This is based upon the principle that the mobile agent is dispatched to each agent server in the itinerary. If the mobile agent is lost due to an agent server crash then a replica is sent to an alternative agent server. The use of temporal replication introduces a number of issues, the most significant being the potential use for adopting the scheme for a group of collaborating mobile agents. Unlike spatial replication techniques [DeAssisSilva00, Pleisch03, Schneider97, Strasser99] there is no need to dispatch replica mobile agents to a static group of equivalent agent servers for each stage of the itinerary. Instead, the mobile shadow exception handling scheme uses a shadow mobile agent to monitor the availability of the application mobile agent at each stage of the itinerary. Consequently, only the set of agent servers specified in the itinerary are visited. This eases complexity since, for each agent server visited in the itinerary, there is no need for the application developer to determine a set of equivalent agent servers to visit in the event of a crash failure.

There are a number of possible extensions to the mobile shadow exception handling scheme, such as extending the failure model and providing a dynamic itinerary that allows the mobile agent to select an alternative agent server in the event of a crash failure. However, this has been considered for future work. Instead, focus was given to an exception handling model for mobile agents, with the aim of allowing exception handling for mobile agents to be better understood, particularly with respect to a group of mobile agents.

2 Research summary

This research has presented a number of issues relating to exception handling and fault tolerance for mobile agents against agent server crash failures. The primary contributions can be considered to be:

- An exception handling model that highlights the key differences for exception handling in mobile agent systems, as opposed to traditional distributed systems.
- A failure model for mobile agent systems.
- A demonstration of a fault tolerance scheme that uses exception handling and temporal replication to protect mobile agents from agent server crash failures.

The mobile shadow exception handling scheme uses temporal replication to protect mobile agents against agent server crash failures. A sequential itinerary is used to determine the boundary of agent servers visited by the mobile agent. This is assumed in other systems that protect mobile agents from agent server crash failures. A replica is spawned before the application mobile agent migrates to the next agent server in the itinerary. Exception handlers are provided for the crash of the replica and application mobile agent. The crash of a replica is transparent to the application developer. In this case the exception handler inspects the sequential itinerary to determine an available agent server that can be occupied by a replacement replica. If the application mobile agent is lost due to an agent server crash failure then the exception handler uses the replica as a replacement. Subsequently, a new replica is spawned to monitor the replacement application mobile agent.

The itinerary is particularly important for the protection of mobile agents against agent server crash failures. The mobile shadow exception handling scheme uses a sequential itinerary to conveniently provide a set of agent servers, available for occupation by one or more replica mobile agents. Furthermore, a natural boundary for error confinement is established. Consequently, the itinerary can provide a boundary for atomic actions in mobile agent systems. The concept of atomic actions is presented in chapter 3.

An itinerary has equally been used in spatial replication mechanisms to establish the set of agent servers occupied by replicas at each stage of the itinerary. However, this has the disadvantage that there is an increase in the state of the mobile agent. For example, entries are required for each stage in the itinerary to represent the alternative agent servers that can be visited in the event of a crash.

3 Criteria for success

With respect to the exception handling model (outlined in chapter 4, section 2) and the mobile shadow exception handling scheme, it is possible to review the introductory chapter to summarise what has been achieved. The criteria for success were presented in chapter 1, section 5. These will now be examined in order to demonstrate the extent to which they have been achieved within the thesis.

a) Create an exception handling model for mobile agent systems

This research has proposed an exception handling model for mobile agents in chapter 4, section 2. Mobile agents execute an itinerary by visiting remote agent servers to interact with software services on the behalf of the user. If a mobile agent receives an exception from a service at a remote agent server then it can retry the service, migrate to an agent server that offers an equivalent service or report back to the home agent server.

Exception handling in mobile agent applications is recursive. Software services may dispatch a mobile agent to satisfy a service request. Similarly, a visiting mobile agent may dispatch a child to perform a task on its behalf for the duration of its visit. Exception handling in mobile agent systems is complex when a mobile agent dispatches a child to perform a task on its behalf. If a child encounters an exception from a service at a remote agent server then the parent must be notified. This is provided that a synchronous relationship exists between parent and child. Consequently, in these circumstances the parent must remain stationary until the child returns.

b) Identify a failure model for mobile agent systems

A failure model was introduced in chapter 4, section 3, for mobile agent systems in general. There are few failure models in existence for mobile agents. The key classifications for mobile agent execution failure are: security, communication, software faults and agent server crashes.

Chapter 4 presents a failure model that summarises, for each class of failure, the scenarios and conditions of failure. The mobile shadow exception handling scheme addresses the crash failure classification for information retrieval applications. The conditions of agent server crash failures have been outlined for information retrieval applications. Namely, these are that network partitions, host crashes and communication link failures eventually recover. Furthermore, remote agent servers are not expected to provide persistency of mobile agents.

c) Development of an exception handling scheme for the protection of mobile agents against agent server crashes

The implementation of an exception handling scheme for the protection of mobile agents against agent server crashes was presented in chapter 4, section 4. The demonstration of the mobile shadow exception handling scheme, using the case study application that is presented in chapter 5, raised significant issues with respect to handling crash failures for mobile agent systems. Firstly, a mobile agent's itinerary is useful to establish a boundary for error confinement. Secondly, a log is required of the agent servers where a mobile agent failed to complete execution due to crash failures. Solutions for the exception handler include: resend the mobile agent, in the event that the agent server eventually recovers, or dispatch the mobile agent to the home agent server for application recovery.

A further aspect identified is the difficulty of protecting children from crash failures. Most of the systems presented in the feature analysis in chapter 6 assume that children are dispatched to perform a task that is independent of the parent. Two of the systems included in the feature analysis provide recovery where a synchronous relationship is introduced between parent and children. However, both are conceptual with no actual implementation for the scenario.

d) Identify the best approach for mobile agent groups to survive agent server crashes

The thesis has concluded that a temporal replication mechanism is preferred for protecting a mobile agent that is a member of a group, from agent server crash failures. Furthermore, agent servers visited in the itinerary can be used to host a replica that replaces the application mobile agent in the event of an agent server crash. This has been demonstrated by the case study application in chapter 5 and the evaluation results in chapter 6.

A significant advantage of the mobile shadow exception handling scheme, with respect to mobile agent groups, is that a replica dynamically migrates with the application mobile agent at each stage of the itinerary. This reduces the performance overheads when compared to spatial replication mechanisms. Spatial replication mechanisms [DeAssisSilva00, Pleisch03, Schneider97, Strasser99] adopt a voting protocol in the event of an agent server crash to elect a new leader from a group of replicas. Furthermore, spatial replication mechanisms produce a larger itinerary, since for each stage in the itinerary a mobile agent replica must be dispatched to n agent servers that are capable of executing the mobile agent. Complexity is therefore reduced by the mobile shadow exception handling scheme, since a group of replicas is not required for each group member. Consequently, each member of the mobile agent group is a single fault tolerant entity.

Re-examination of these criteria, with reference to the relevant parts of the thesis, has shown that this research has been successful. The research has addressed the overall aim of developing and evaluating a fault tolerance protocol that uses exception handling to protect mobile agents from agent server crash failures. Furthermore, this thesis adds to current fault tolerance research for mobile agent systems.

4 Future work

There are many further directions for the research that involve extending the implementation of the mobile shadow exception handling scheme, extending the failure model and employing the mobile shadow exception handling scheme for a group of collaborating mobile agents. This section examines these areas in further detail.

4.1 Implementation

The implementation of the mobile shadow exception handling scheme could be extended in the following ways. Firstly, in the event of an agent server crash a replica mobile agent, or shadow, skips the crashed agent server and executes at the next available agent server in the itinerary. The Ajanta [Tripathi02] mobile agent system provides an itinerary pattern to allow a mobile agent to log failed visits to agent servers. When a mobile agent completes its trip the home agent server has the following options:

- Dispatch the mobile agent to remote agent servers to retry execution.
- Dispatch the mobile agent to an alternative agent server that provides an equivalent service.

Most mobile agent systems require that the application developer is aware of the agent servers that offer the required service. With the introduction of directory services it is possible to dynamically locate a software service that meets functional requirements. In the event of an agent server crash, fault tolerance schemes [DeAssisSilva00, Pleisch03, Schneider97, Strasser99] for mobile agents provide the option to dispatch a replica mobile agent to an alternative agent server that provides an equivalent service. Consequently, the mobile shadow exception handling scheme presented in chapter 4 could be extended to include a directory service that enables mobile agents to lookup agent servers that offer required services. This facility is already provided in systems that implement the FIPA [Fipa04] and MASIF [Milojivcic98] standards.

An alternative approach [DeAssisSilva00, Pleisch03, Schneider97, Strasser99] to the mobile shadow exception handling scheme replicates a mobile agent, at each stage of its itinerary, to a set of agent servers that provide the desired service. If a mobile agent is lost due to an agent server crash, a voting algorithm is run by the replicas to elect a new leader. There may be

savings in recovery performance since a replica is immediately available at an alternative agent server. However, overheads still exist for electing a new leader and sending replicas to redundant agent servers at each stage of the itinerary. It would be interesting to investigate further the performance of this approach with the mobile shadow exception handling scheme.

4.2 Extended failure model

This thesis has constrained the design of the mobile shadow exception handling scheme to information retrieval applications, i.e. visiting mobile agents consume information at remote agent servers. A failure model and conceptual design was presented in chapter 4. The next logical step is to apply the mobile shadow exception handling scheme to applications where mobile agents can modify the state of remote agent servers through interaction with software resources. This in turn implies transactions to rollback the state of a mobile agent to arrival at the agent server that crashed. Imposing this additional assumption has the following implications for future work:

- Investigate a distributed transaction model that establishes a boundary of mobile agent execution. Furthermore, the transaction model identifies transaction mechanisms and the party responsible for mobile agent persistency. Possibilities include the home agent server or remote agent servers visited.
- Create an XML schema to specify the workflow of a mobile agent. The workflow describes the agent servers visited by a mobile agent and alternative agent servers that provide an equivalent service in the event of a crash failure. Furthermore, meta-information can be included that describes the activities of the mobile agent. For example, is it possible to compensate or retry the activity of a mobile agent?
- Introduce a heterogeneous mechanism, e.g. XML, to save the state of mobile agents into stable storage.

4.3 Exception handling for mobile agent groups

This thesis has presented the mobile shadow exception handling scheme that allows a single mobile agent to survive agent server crash failures. The mobile shadow exception handling scheme migrates with the application mobile agent for protection against agent server crash failures in information retrieval environments. The next logical step is to investigate the use of the mobile shadow exception handling scheme for a group of collaborating mobile agents. This is expected to be performed using a case study application. Four scenarios are envisaged for mobile agent group collaboration:

1. **Preserve bandwidth savings for mobile agents with large itineraries:** A large itinerary that encompasses hosts that offer discrete services produces a "fat" mobile agent. The mobile agent must encompass knowledge about the actions to perform at each agent server. Furthermore, a greater amount of knowledge is required to process

the information obtained from different software services at remote agent servers. Consequently, there is a danger that the savings in bandwidth gained from employing mobile agent technology is sacrificed. Furthermore, the code of the mobile agent is monolithic resulting in complex maintainability. To preserve the bandwidth savings and reduce complexity it is natural to introduce a hierarchy of "lean" mobile agents that serve to decompose the workflow. Consequently, collaboration must be introduced to synchronise information for workflow interdependencies between mobile agents in the hierarchy.

2. **Collaboration to preserve shared application semantics:** A group workflow structure of mobile agents may share common application semantics. For example, a group of e-commerce mobile agents may share application constraints such as budget limit, delivery date and possibly location. If one of the semantics is violated, then all agents must be informed to enable alternative planning for purchases. This collaboration is application specific.
3. **Collaboration to prevent duplicate migration:** Mobile agents can be used to traverse large web databases, e.g. searching internet databases of web pages [Cabri00]. One use is to employ mobile agents to traverse links and form a global perspective of the hyperlink structure. A clone may be deployed to follow new hyperlinks to databases or retrieve new material. Cloning is much more efficient as opposed to using a single monolithic agent since the clone is a leaner agent, i.e. it employs a smaller itinerary and has no accumulated state upon creation. To avoid duplicate searches between clones collaboration is necessary, e.g. a marker could be installed at a site to inform clones that the site has been visited and searched on a specific date and time. Future clones that visit the site may only search the database for those entries after a specific date/time, avoiding redundant searching and processing.
4. **Collaboration for mobile computing:** Single hop mobile agent technology is useful for the mobile computing domain, e.g. e-auctions or e-conferencing. A mobile agent could be dispatched from a mobile computer to negotiate at a central server. The mobile agent negotiates with other mobile agent representatives on the behalf of its disconnected user. This scenario is useful for low powered mobile devices that suffer frequent disconnections.

Scenarios 1, 2 and 3 are of particular interest where shared application semantics introduce dependencies between collaborating mobile agents. The following conceptual model is foreseen for a group of collaborating mobile agents. A group has a global application state, visible to all members, that is limited by a set of shared application constraints, i.e. *constraints* = { C_1, C_2 ... C_c }. Actions that are performed by a member at an agent server AG_k modify the group application state.

The exceptions raised in a group are classified as *local* and *group*. Local exceptions are private to each member and handled autonomously. Each member has a set of local exceptions, $local = \{ le_1, le_2, \ldots le_n \}$ and corresponding handlers, $internal\_handlers = \{ lh_1, lh_2, \ldots lh_n \}$. A group exception ge_a is raised by a member when a shared application constraint C_a is violated. Furthermore, a group exception event is described by the triple, $ge = \{ ID, AG_k, name \}$, where *ID* is the identifier of the member that produced the exception, AG_k is the agent server where the member identified by *ID* executed and *name* is a unique name for the exception. Group exceptions, $group\_exceptions = \{ge_1, ge_2, \ldots ge_c \}$, are known to all members, i.e. members are aware of the group exceptions that can be raised and all members are notified of a group exception occurrence. Each member has a corresponding set of group exception handlers, i.e. $group\_handlers = \{ gh_1, gh_2, \ldots gh_c \}$. When a member is notified of a group exception ge_a the corresponding handler gh_a is invoked.

A CA action framework [Xu00b], introduced in chapter 3, is used in traditional distributed applications to confine errors and exception handling within a group of distributed participants. For mobile agents a boundary for error confinement could include:

- Mobile agents that are members of the group.
- Software resources at remote agent servers modified by interaction with mobile agents.

Adopting a CA action framework [Xu00b] for mobile agents is significantly challenging with respect to the mobility of group members. An important problem to address is notifying all group members of an exception occurrence. In any distributed system there is a communication delay introduced for sending and receiving messages. However, with mobile agents this delay is likely to be increased due to the ability of mobile agents to dynamically relocate between remote hosts. This raises significant concerns when notifying group participants of exception occurrences and changes of the shared application state. Consequently, a mechanism is needed to communicate exceptions and changes in the group application state. One possibility would be to investigate the use of tuple space communication for mobile agents [Cabri02, Omicini01, Murphy01], introduced in chapter 2, section 2.5. Mobile agents can communicate by inserting, modifying and deleting objects at a shared memory space provided at each agent server. A tuple space communication mechanism offers the advantage that mobile agents do not have to synchronise location for communication. This concept may be combined with gossip protocols [Ganesh03, Gupta01, Ranganathan01] to disseminate each member's knowledge to the rest of the group. Eventually, all members learn of exceptions and application state changes in the group. However, further research is required for adopting gossip protocols [Ganesh03, Gupta01, Ranganathan01] for mobile agents. This is necessary to ensure that the extra bandwidth is minimised to maintain the benefit of using mobile agent technology.

5 Summary

This thesis has examined the fundamental principles and challenges for mobile agents to survive agent server and host crash failures. A new approach is then proposed that has the potential to be adopted in information retrieval applications. The new approach, called the mobile shadow exception handling scheme, is described in detail. This includes the use of a case study application to demonstrate the principle concepts of the mobile shadow exception handling scheme. The mobile shadow exception handling scheme reduces complexity for a group of mobile agents to survive server crashes. Each group member is a single fault tolerant entity with respect to server crash failures. The thesis has highlighted the importance of reducing the complexity introduced through fault tolerance, in order to preserve the potential bandwidth savings gained from using the mobile agent paradigm. An evaluation of the mobile shadow exception handling scheme identified its relative merits and areas of future work have been identified for ways in which the research could be carried forward.

References

[Acharya97] A.Acharya, M.Ranganathan and J.Saltz, "Sumatra: A Language for Resource-Aware Mobile Programs," in *Mobile Object Systems: Towards the Programmable Internet*, pp.111-130, Springer Verlag, 1997.

[Adobe85] Adobe Systems Inc, Addison-Wesley, *Postscript Language Reference Manual*, 1985.

[Anderson81] T.Anderson and P.A.Lee, *Fault Tolerance Principles and Practice.* Prentice-Hall International, 1981.

[Aridor98] Y.Aridor and D.B.Lange, "Agent Design Patterns: Elements of Agent Application Design," in *Proceedings of the 2nd International Conference on Autonomous Agents*, Minneapolis, U.S.A., May, 1998, pp.108-115.

[Baeumer03] C.Baeumer, M.Breugst, S.Chay and T.Magedanz, *Grasshopper - A Universal Agent Platform Based on OMG MASIF and FIPA Standards.* http://213.160.69.23/grasshopper-website/links.html (September 2003)

[Beans04] *Jumping Beans.* http://www.jumpingbeans.com (September 04)

[Bellifemine99] F.Bellifemine, A.Poggi and G.Rimassa, "JADE - A FIPA-Compliant Agent Framework," in *Proceedings of the 4th International Conference on The Practical Application of Intelligent Agents and Multi-Agent Technology*, London, U.K., April, 1999, pp.97-108.

[Binder01] W.Binder, "Design and Implementation of the JSEAL2 Mobile Agent Kernel," in *Proceedings of the 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, U.S.A., January, 2001, pp.35-47.

[Boggs73] J.K.Boggs, "IBM Remote Job Entry Facility: Generalise Subsystem Remote Job Entry Facility," Technical Report, IBM Technical Disclosure Bulletin 752, IBM, August, 1973.

[Braun01] P.Braun, J.Eismann, C.Erfurth and W.R.Rossak, "Tracy - A Prototype of an Architected Middleware to Support Mobile Agents," in *Proceedings of the 8th IEEE Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Washington D.C., U.S.A., April, 2001, pp.255-260.

[Brazier02] F.M.T.Brazier, B.J.Overeinder, M. van Steen and N.J.E.Wjingaards. "Agent Factory: Generative Migration of Mobile Agents in Heterogeneous Environments," in *Proceedings of the ACM Symposium on Applied Computing (SAC2002)*, Madrid, Spain, March, 2002, pp.101-106.

[Cabri00] G.Cabri, L.Leonardi and F.Zambonelli, "Mars: A Programmable Coordination Architecture for Mobile Agents," *IEEE Internet Computing*, 4(4), pp.26-35, 2000.

[Cabri02] G.Cabri, L.Leonardi and F.Zambonelli, "Engineering Mobile Agent Applications via Context-Dependent Coordination," *IEEE Transactions on Software Engineering*, 28(11), pp.1039-1055, 2002.

[Campadello00] S.Campadello, H.Helin, O.Koskimies, P.Misikangas, M.Makela and K.Raatikainen, "Using Mobile Agents and Intelligent Agents to Support Nomadic Users," in *Proceedings of the 6th International Conference on Intelligence in Networks (ICIN2000)*, Bordeaux, France, January, 2000.

[Campbell79] R.H.Campbell, K.H.Horton and G.G.Bedford, "Simulations of a Fault Tolerant Deadline Mechanism," in *Proceedings of the 9th International Symposium on Fault Tolerant Computing Systems (FTCS-9)*, Madison, U.S.A., June, 1979, pp.95-101.

[Campbell86] R.H.Campbell and B.Randell, "Error Recovery in Asynchronous Systems," *IEEE Transactions on Software Engineering*, 12(8), pp.811-826, 1986.

[Cao03] J.Cao, Y.Sun, Y.Wang and S.K.Das, "Scalable Load Balancing on Distributed Web Servers Using Mobile Agents," *Journal of Parallel and Distributed Computing*, 63(10), pp.996-1005, 2003.

[Cardelli97] L.Cardelli, "Mobile Computation," in *Mobile Object Systems Towards the Programmable Internet*, pp.4-6, Springer Verlag, 1997.

[Chen78] L.Chen and A.Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Proceedings of the 8th International Symposium on Fault-Tolerant Computing Systems (FTCS-8)*, Toulouse, France, June, 1978, pp.3-9.

[Chess95] D.M.Chess, C.G.Harrison and A.Kershenbaum, "Mobile Agents: Are they a Good Idea?," Technical Report, IBM Research Division, http://www.research.ibm.com/iagents/publications.html, 1995.

[Coabs04] *Control of Agent Based Systems*. http://coabs.globalinfotek.com (June2004)

[Dasgupta99] P.Dasgupta, N.Narasimhan, L.Moser and P.M.Melliar-Smith, "MAgNET: Mobile Agents for Networked Electronic Trading," *IEEE Transactions on Knowledge and Data Engineering*, 24(6), pp.509-525, 1999.

[DeAssisSilva00] F.M. de Assis Silva and R.Popescu-Zeletin, "Mobile Agent-Based Transactions in Open Environments," *IEICE Transactions on Communications*, E83-B(5), pp.973-987, 2000.

[Dennett87] D.C.Dennett, *The Intentional Stance*. MIT Press, 1987.

[Emmerich00] W.Emmerich, C.Mascolo and A.Finkelstein, "Implementing Incremental Code Migration with XML," in *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June, 2000, pp.397-406.

[Fipa04] *Fipa*. http://www.fipa.org (June 2004)

[Franklin97] S.Franklin and A.Graesser, "Is it an Agent or Just a Program? A Taxonomy for Autonomous Agents," in *Intelligent Agents III Agent Theories, Architectures and Languages*, pp.21-35, Springer Verlag, 1997.

[Fuggetta98] A.Fuggetta, G.P.Picco and G.Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering*, 24(5), pp.342-361, 1998.

[Ganesh03] A.J.Ganesh, A.Kermarrec and L.Massoulie, "Peer-to-Peer Membership Management for Gossip-Based Protocols," *IEEE Transactions on Computers*, 52(2), pp.139-149, 2003.

[Garcia01] A.F.Garcia, C.M.F.Rubira, A.Romanovsky and J.Xu, "A Comparitive Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software," *Journal of Systems Software,* 59(2), pp.197-222, 2001.

[Gelernter85] D.Gelernter, "Generative Communication in Linda," *ACM Computing Surveys*, 7(1), pp.80-112, 1985.

[Genesereth94] M.R.Genesereth and S.P.Ketchpel, "Software Agents," *Communications of the ACM*, 37(7), pp.48-53, 1994.

[Goulouris00] G.Goulouris, J.Dollimore and T.Kindburg, *Distributed Systems Concepts and Design*. Addison Wesley, 3rd ed., 2000.

[Gray02] R.S.Gray, G.Cybenko, D.Kotz, R.A.Peterson and D.Rus, "D'Agents: Applications and Performance of a Mobile-Agent System," *Software Practice and Experience,* 32(6), pp.543-573, 2002.

[Grey00] D.J.Grey, P.Dunne and R.I.Ferguson, "On Searching the WWW with Mobile Collaborative Agents," in *Proceedings of the IIIS International Conference on Systems, Analysis and Synthesis (SCI-ISAS2000)*, Orlando, U.S.A., July, 2000, pp.59-64.

[Grimstrup02] A.Grimstrup, R.Gray, D.Kotz, M.Breedy, M.Carvalho, T.Cowin, D.Chacon, J.Barton, C.Garrett and M.Hofmann, "Toward Interoperability of Mobile-Agent Systems," in *Proceedings of the Sixth IEEE International Conference on Mobile Agents*, Barcelona, Spain, October, 2002, pp.106-120.

[Gupta01] I.Gupta, R. van Renesse and K.P.Birman, "Scalable Fault Tolerant Aggregation in Large Process Groups," in *Proceedings of the International Conference on Dependable Systems and Networks (DNS'01)*, Goteborg, Sweden, July, 2001, pp.433-442.

[Hadzilacos94] V.Hadzilacos and S.Toueg, "A Modular Approach to Fault Tolerant Broadcasts and Related Problems," Technical Report TR94-1425, Dept. Computer Science, Cornell University, Ithaca, U.S.A., May 1994.

[Hermann00] K.Hermann and M.Zopf, *Ametas White Paper Series*, http://www.vsb.cs.uni-frankfurt.de/ametas/docs/white/AllWhite.pdf, 2000.

[Holder99] O.Holder, I.Ben-Shaul and H.Gazit, "Dynamic Layout of Distributed Applications in FarGo," in *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, U.S.A., May, 1999, pp.163-173.

[Horning74] J.J.Horning, H.C.Lauer, P.M.Melliar-Smith and B.Randell, "A Program Structure for Error Detection and Recovery," in *Proceedings of the International Symposium on Operating Systems*, Rocquencourt, France, April, 1974, pp.171-187.

[Huhns97] M.N.Huhns and M.P.Singh, "*Readings In Agents*," Chapter 1, pp.1-23, Morgan Kauffmann, 1997.

[Jalote86] P.Jalote and R.H.Campbell, "Atomic Actions for Fault-Tolerance Using CSP," *IEEE Transactions on Software Engineering*, 12(1), pp.59-68, 1986.

[Jalote94] P.Jalote, *Fault Tolerance in Distributed Systems*. Prentice-Hall International, 1994.

[JavaSpaces03] *JavaSpaces v2.0 Specification*. http://www.sun.com/jini/specs/js2_0.pdf (June2003)

[Johansen99] D.Johansen, K.Marzullo, F.Schneider, K.Jacobsen and D.Zagorodnov, "NAP:Practical Fault Tolerance for Itinerant Computations," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Texas, U.S.A., June, 1999, pp.180-189.

[Johansen02] D.Johansen, K.Lauvset, R. van Renesse, F.Schneider, N.Sudmann and K.Jacobsen, "A TACOMA Retrospective," *Software Practice and Experience*, 32(6), pp.605-619, 2002.

[Kienzle01] J.Kienzle, A.Romanovsky and A.Stroheimer, "Open Multithreaded Transactions Keeping Threads and Exceptions Under Control," in *6th International Workshop on Object-Oriented Real-Time Dependable Systems*, Roma, Italy, January, 2001, pp.209-217.

[Kim84] K.H.Kim, "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults," in *Proceedings of the 4th International Conference on Distributed Computing Systems*, San Francisco, U.S.A., May, 1984, pp.577-632.

[Klein99] M.Klein and C.Dallarocus, "Exception Handling in Agent Systems," in *Proceedings of the 3rd International Conference on Autonomous Agents (Agent 99)*, Seattle, U.S.A., May, 1999, pp.62-68.

[Kotz99] D.Kotz and R.S.Gray, "Mobile Agents and the Future of the Internet," *ACM Operating Systems Review*, 33(3), pp.7-13, 1999.

[Kotz02] D.Kotz, R.Gray and D.Rus, *Future Directions for Mobile Agent Research*, http://dsonline.computer.org/0208/f/kot_print.htm, (November 2002)

[Lange99] D.B.Lange and M.Oshima, "Seven Good Reasons for Mobile Agents," *Communications of the ACM*, 42(3), pp.88-89, 1999.

[Laprie85] J.C.Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," in *Proceedings of the 15th International Symposium on Fault Tolerant Computing Systems*, Michigan, U.S.A., June, 1985, pp.2-11.

[Laprie92] J.C.Laprie, *Dependability: Basic Concepts and Terminology – In English, French, German and Japanese*. Springer Verlag, 1992.

[Lehman99] T.J.Lehman, S.W.McLaughry and P.Wycoff, "TSpaces: The Next Wave," in *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS-32)*, Maui, Hawaii, January, 1999.

[Levy88] H.Levy, E.Jul, N.Hutchinson and A.Black, "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, 6(2), pp.109-133, 1988.

[Lieberman95] H.Lieberman, "Letizia: An Agent that Assists Web Browsing," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (ICJAI-95)*, Montreal, Canada, August, 1995, pp.924-929.

[Lingau95] A.Lingau, O.Drobinky and P.Domel, "A http Infrastructure for Mobile Agents," in *Proceedings of the 4th International WWW Conference*, Boston, U.S.A., April, 1995, pp.23-32.

[Liu02] J.Liu, Q.Zhang, B.Li, W.Zhu and J.Zhang, "A Unified Framework for Resource Discovery and QoS Aware Provider Selection in Ad-hoc Networks," *ACM Mobile Computing and Communications Review*, 6(1), pp.13-21, 2002.

[Maes95] P.Maes, "Intelligent Software," *Scientific American*, 273(3), pp.84-86, 1995.

[Magnin02] L.Magnin, V.T.Pham, A.Dury and N.Besson, "Our Guest Agents are Welcome to Your Agent Platforms," in *Proceedings of the ACM Symposium on Applied Computing (SAC2002)*, Madrid, Spain, March, 2002, pp.101-106.

[Marsden02] E.Marsden, J.Fabre and J.Arlat, "Dependability of CORBA Systems: Service Characterization by Fault Injection," in *Proceedings of the. 21st IEEE Symposium on Reliable Distributed Systems*, Suita, Japan, October, 2002, pp.276-285.

[Marwaha02] S.Marwaha, C.H.Tham and D.Srinivasan, "Mobile Agents Based Routing Protocol for Mobile Ad Hoc Networks," in *Proceedings of the IEEE Globecom 2002: The World Converges*, Taipei, Taiwan, November, 2002.

[Milojivcic98] D.Milojivcic, M.Breugst, I.Busse, J.Campbell, S.Covaci, B.Friedman, K.Kosaka, D.Lange, K.Ono, M.Oshima, C.Tham, S.Virdhagriswaran and J.White, "MASIF The OMG Mobile Agent System Interoperability Facility," in *Proceedings of the 2nd International Workshop on Mobile Agents (MA '98)*, Stuttgart, Germany, September, 1998, pp.50-67.

[Milojivcic99] D.Milojivcic, "Trend Wars: Mobile Agent Applications," *IEEE Concurrency*, 7(3), pp.80-90, 1999.

[Minar99] N.Minar, K.H.Kramer and P.Maes, "Cooperating Mobile Agents for Dynamic Network Routing," in *Software Agents for Future Communication Systems*, pp. 287-304, Springer Verlag, 1999.

[Mingas03] N.Mingas, W.J.Buchanan and K.A.McArtney, "Mobile Agents for Routing, Topology Discovery and Automatic Network Reconfiguration in Ad Hoc Networks," in *Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'03)*, Huntsville, U.S.A., April, 2003, p200-206.

[Misikangas00] P.Misikangas and K.Raatikainen, "Agent Migration Between Incompatible Agent Platforms," in *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, Taipei, Taiwan, April, 2000, pp.4-10.

[Mohindra00] A.Mohindra, A.Purakayastha and P.Tahiti, "Exploiting Non-Determinism for Reliability of Mobile Agent Systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, New York, U.S.A., June, 2000, pp.144-153.

[Muller96] J.P.Muller, *The Design of Intelligent Agents: A Layered Approach*, Springer Verlag, 1996.

[Murphy99] A.L.Murphy and G.P.Picco, "Reliable Communication for Highly Mobile Agents," in *Proceedings of the 3rd International Symposium on Mobile Agents (MA'99)*, California, U.S.A., October, 1999, pp.141-150.

[Murphy01] A.L.Murphy, G.P.Picco and G.C.Roman, "LIME: A Middleware for Physical and Logical Mobility," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDS-21)*, Phoenix, U.S.A., 2001, pp.524-536.

[Ndumu97] D.T.Ndumu and H.S.Nwana, "Research and Development Challenges for Agent-Based Systems," *IEE Proceedings on Software Engineering*, 144(1), pp.2-10, 1997.

[Nuttall94] M.Nuttall, "A Brief Survey of Systems Providing Process or Object Migration Facilities," *ACM Operating Systems Review*, 28(4), pp.64-80, 1994.

[Nwana96] H.S.Nwana, "Software Agents: An Overview," *The Knowledge Engineering Review*, 11(3), pp.205-244, 1996.

[Nwana99] H.S.Nwana and D.T.Nolumn, "A Perspective on Software Agents Research," *Knowledge Engineering Review*, 14(2), pp.125-142, 1999.

[Omicini01] A.Omicini and E.Denti, "From Tuple Spaces to Tuple Centres," *Science of Computer Programming*, 41(3), p277-294, 2001.

[Oshima98] M.Oshima, G.Karjoth and K.Ono, *Aglets Specification 1.1 Draft.* http://www.trl.ibm.co.jp/aglets/spec11.html, 1998.

[Parnas90] D.L.Parnas, J.Scouwen and K.S.Po, "Evaluation of Safety-Critical Software," *Communications of the ACM*, 33(6), pp.636-648, 1990.

[Pears03] S.Pears, J.Xu and C.Boldyreff, "Mobile Agent Fault Tolerance for Information Retrieval Applications: An Exception Handling Approach," in *Proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS'03)*, Pisa, Italy, April, 2003, pp.115-124.

[Pears03b] S.Pears, J.Xu and C.Boldyreff, "A Dynamic Shadow Approach for Mobile Agents to Survive Crash Failures," in *Proceedings of the 6th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, Hokkaido, Japan, May, 2003, pp.113-120.

[Peine02] H.Peine, "Application and Programming Experience with the Ara Mobile Agent System," *Software Practice and Experience*, 32(6), pp.515-541, 2002.

[Perkins94] C.E.Perkins and P.Bhagwat, "Highly Dynamic Destination Sequenced Vector Routing (DSDV) for Mobile Computers," in *Proceedings of the ACM Conference on Communications Architectures, Protocols and Applications (SIGCOMM'94)*, London, U.K., August, 1994, pp.234-244.

[Perkins99] C.E.Perkins, E.M.Rayer and S.R.Das, "Ad Hoc On Demand Distance Vector (ASDV) Routing," in *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, U.S.A., February, 1999, pp.90-100.

[Perry92] D.E.Perry and A.L.Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, 17(4), pp.40-52, 1992.

[Picco98] G.P.Picco, "μCode: A Lightweight and Flexible Mobile Code Toolkit," in *Proceedings of the 2nd International Workshop on Mobile Agents 98 (MA '98)*, Stuttgart, Germany, September, 1998, pp.160-171.

[Picco98b] G.P.Picco and M.Baldi, "Evaluating the Tradeoffs of Mobile Code Design Paradigms on Network Management," in *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April, 1998, pp.146-155.

[Picco01] G.P.Picco, "Mobile Agents: An Introduction," *Journal of Microprocessors and Microsystems*, 25(2), pp.65-74, 2001.

[Pinsdorf02] U.Pinsdorf and V.Roth, "Mobile Agent Interoperability Patterns and Practice," in *Proceedings of the 9th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2002)*, Lund, Sweden, April, 2002, pp.238-244.

[Pleisch00] S.Pleisch and A.Schiper, "Modeling Fault-Tolerant Mobile Agent Execution As a Sequence of Agreement Problems," in *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, Piscataway, U.S.A., October, 2000, pp.11-20.

[Pleisch01] S.Pleisch and A.Schiper, "FATOMAS - A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*, Goteborg, Sweden, July, 2001, pp.215-224.

[Pleisch03] S.Pleisch and A.Schiper, "Fault-Tolerant Mobile Agent Execution," *IEEE Transactions on Computers*, 52(2), pp.209-222, 2003.

[Randell75] B.Randell, "Systems Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, 1(2), pp.220-232, 1975.

[Ranganathan01] S.Ranganathan, A.D.George, R.W.Todd and M.C.Chidester, "Gossip Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters," *Cluster Computing*, 4(3), pp.197-209, 2001.

[Rao95] A.S.Rao and M.P.Georgeff, "Bdi Agents: From Theory To Practice," in *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95)*, San Francisco, U.S.A., June, 1995, pp.312-319.

[Romanovsky00] A.Romanovsky, "Extending Conventional Languages by Distributed/Concurrent Exception Resolution," *Journal of Systems Architecture*, 46(1), pp.79-95, 2000.

[Roth01] V.Roth and M.Jalali, "Concepts and Architecture of a Security-Centric Mobile Agent Server," in *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, Dallas, U.S.A., March, 2001, pp.435-442.

[RoyChoudhury00] R.RoyChoudhury, S.Bandyopadhyay and K.Paul, "A Distributed Mechanism for Topology Discovery in Ad Hoc Wireless Networks Using Mobile Agents," in *Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking and Computing*, Boston, U.S.A., 2000, pp.145-146.

[Sakamoto00] T.Sakamoto, T.Sekiguchi and A.Yonezawa, "Bytecode Transformation for Portable Thread Migration in Java," in *Agent Systems, Mobile Agents and Applications, 2nd International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA 2000)*, pp.16-28, Springer Verlag, 2000.

[Schlichting83] R.Schlichting and F.Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems*, 1(3), pp.222-238, 1983.

[Schneider97] F.Schneider, "Towards Fault-Tolerant and Secure Agentry," in *Proceedings of the 11th International Workshop on Distributed Algorithms*, Saarbrucken, Germany, September, 1997, pp.1-14.

[Schoder00] D.Schoder and T.Eymann, "The Real Challenges of Mobile Agents," *Communications of the ACM*, 43(6), pp.111-112, 2000.

[Sekiguchi99] T.Sekiguchi, H.Masuhara and A.Yonezawa, "A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation," in *Coordination Languages and Models*, pp.211-226, Springer Verlag, 1999.

[Shaw95] M.Shaw, "Abstraction to Software Architecture Perspectives and Tools to Support Them," *IEEE Transactions on Software Engineering*, 21(4), pp.1-44, 1995.

[Shoham93] Y.Shoham, "Agent-Oriented Programming," *Artificial Intelligence*, 60(1), pp.51-92, 1993.

[Silva00] L.Silva, V.Batista and J.Silva, "Fault-Tolerant Execution of Mobile Agents," in *Proceedings of the International Conference on Dependable Systems and Networks*, New York, U.S.A., June, 2000, pp.135-143.

[Strasser98] M.Strasser and K.Rothermel, "Reliability Concepts for Mobile Agents," *International Journal of Co-operative Information Systems*, 7(4), pp.355-382, 1998.

[Strasser99] M.Strasser, J.Baumann and M.Schwehm, "An Agent-Based Framework for the Transparent Distribution of Computations," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, U.S.A., June, 1999, pp.376-382.

[Suri00] N.Suri, J.M.Bradshaw, M.R.Breedy, P.T.Groth, G.A.Hill and R.Jeffers, "Strong Mobility and Fine Grained Resource Control in NOMADS," in *LNCS 1882:Proceedings of the 2nd International Symposium on Agent Systems and Applications and 4th International Symposium on Mobile Agents (ASA/MA 2000)*, pp.2-15, Springer Verlag, 2000.

[Thoma03] Y.Thoma, "Fault Tolerance in Autonomic Computing Environment," in *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real Time Distributed Computing (ISORC'03)*, Hakodate, Japan, May, 2003.

[Tripathi01] A.Tripathi and R.Miller, "Exception Handling in Agent-Oriented Systems," in *LNCS 2022: Advances in Exception Handling Techniques*, pp.128-146, Springer Verlag, 2001.

[Tripathi02] A.Tripathi, N.M.Karnik, T.Ahmed, R.D.Singh, A.Prakash, V.Kakani, M.K.Vora and M.Pathak, "Design of the Ajanta System for Mobile Agent Programming," *Journal of Systems and Software*, 62(2), pp.123-140, 2002.

[Vogler97] H.Vogler, T.Hunklemann and M.Moschgath, "An Approach for Mobile Agent Security and Fault Tolerance Using Distributed Transactions," in *Proceedings of the 1997 International Conference on Parallel and Distributed Systems (ICPADS'97)*, Seoul, Korea, December 1997, pp.268-274.

[Waldo01] J.Waldo, "Mobile Code, Distributed Computing and Agents," *IEEE Intelligent Systems,* 16(2), pp.10-12, 2001.

[Wang01] X.Wang, F.Li, S.Ishihara and T.Mizuni, "A Multi-cast Routing Algorithm Based on Mobile Multicast Agents in Ad-Hoc Networks," *IEICE Transactions on Communications*, E84-B(8), pp.2087-2094, 2001.

[Wong97] D.Wong, N.Paciorek, T.Walsh and J.DiCelie, "Concordia an Infrastructure for Collaborating Mobile Agents," in *Proceedings of the First International Workshop on Mobile Agents (MA'97)*, Berlin, Germany, April, 1997, pp.86-97.

[Wooldridge94] M.Wooldridge and N.Jennings, "Agent Theories, Architectures and Languages: A Survey," in *Intelligent Agents: ECAI-94 Workshop on Agent Theories, Architectures and Languages*, pp.1-39, Springer Verlag, 1994.

[Wooldridge94b] M.Wooldridge and M.Fischer, "Agent Based Software Engineering," Technical Report, Department Of Computation, University Of Manchester, 1994.

[Wooldridge95] M.Wooldridge and N.Jennings, "Intelligent Agents: Theory and Practice," *The Knowledge Engineering Review*, 10(2), pp.115-152, 1995.

[Wooldridge97] M.Wooldridge, "Agent-Based Software Engineering," *IEE Proceedings on Software Engineering*, 144(1), pp.26-37, 1997.

[Xerces04] *Xerces*. http://xml.apache.org/xerces-j/ (July2004)

[Xu95] J.Xu, B.Randell, A.Romanovsky, C.Rubira, R.Stroud and Z.Wu, "Fault Tolerance in Concurrent Object-Oriented Software Through Co-ordinated Error Recovery," in *Proceedings of the 25th IEEE International Symposium on Fault Tolerant Computing*, Pasadena, U.S.A., June, 1995, pp.499-508.

[Xu00] J.Xu and B.Randell, "Tutorial: Exception Handling and Software Fault Tolerance," in *Proceedings of the International Conference on Dependable Systems and Networks*, New York, U.S.A., June, 2000.

[Xu00b] J.Xu, A.Romanovsky and B.Randell, "Concurrent Exception Handling and Resolution in Distributed Object Systems," *IEEE Transactions on Parallel and Distributed Systems*, 11(10), pp.1019-1032, 2000.